Prepublication reviewers say

"This book demystifies the magic of computers. . .it's the machine language masterpiece we've all been waiting for!"

" . an excellent introduction to machine language. .I love the consistent use of both hex and decimal, the overall readability, and the numerous summaries!"

MACHINE • LANGUAGE
for the Commodore 64 and other Commodore computers
• Jim Butterfield •

Now! A comprehensive tutorial that introduces programmers of all levels to the principles of machine language— what it is, how it works, and how to program with it! Based on the author's own intense machine language seminars, this learn-by-doing guide explores machine code in the real environment of Commodore personal computers, examining important concepts such as output   address modes   .linking BASIC and machine language   memory maps of the interface chips   .and much more!!

CONTENTS
First Concepts/Controlling Output/Flags, Logic, And Input/Numbers, Arithmetic, And Subroutines/Address Modes/Linking BASIC And Machine Language/Stack, USR, Interrupt, And Wedge/Timing, Input, Output, And Conclusion

	IMM 2	ZPAG 2	Z,X 2	(I,X) 2	(I),Y 2	ABS 3	A,X 3	A,Y 3
ORA	09	05	15	01	11	0D	1D	19
AND	29	25	35	21	31	2D	3D	39
EOR	49	45	55	41	51	4D	5D	59
ADC	69	65	75	61	71	6D	7D	79
STA		85	95	81	91	8D	9D	99
LDA	A9	A5	B5	A1	B1	AD	BD	B9
CMP	C9	C5	D5	C1	D1	CD	DD	D9
SBC	E9	E5	F5	E1	F1	ED	FD	F9

Op Code ends in −1, −5, −9, or −D

	IMM 2	ZPAG 2	Z,X 2	Z,Y 2	ABS 3	A,X 3	A,Y 3
ASL		06	16		0E	1E	
ROL		26	36		2E	3E	
LSR		46	56		4E	5E	
ROR		66	76		6E	7E	
STX		86		96	8E		
LDX	A2	A6		B6	AE		BE
DEC		C6	D6		CE	DE	
INC		E6	F6		EE	FE	

Op Code ends in −2, −6, or −E

BPL	10	BMI	30
BVC	50	BVS	70
BCC	90	BCS	B0
BNE	D0	BEQ	F0

Branches −0

	ABS	(IND)
JSR	20	
JMP	4C	6C

Jumps

	IMM 2	ZPAG 2	Z,X 2	ABS 3	A,X 3
BIT		24		2C	
STY		84	94	8C	
LDY	A0	A4	B4	AC	BC
CPY	C0	C4		CC	
CPX	E0	E4		EC	

Misc. −0, −4, −C

	0−	1−	2−	3−	4−	5−	6−	7	8−	9−	A−	B−	C−	D−	E−	F−
−0	BRK				RTI		RTS									
−8	PHP	CLC	PLP	SEC	PHA	CLI	PLA	SEI	DEY	TYA	TAY	CLV	INY	CLD	INX	SED
−A	ASL-A		ROL-A		LSR-A		ROR-A		TXA	TXS	TAX	TSX	DEX		NOP	

Single-byte Op Codes −0, −8, −A

MACHINE
LANGUAGE
FOR THE
COMMODORE 64
and Other
Commodore
Computers

James Butterfield

Machine Language for the Commodore 64 and Other Commodore Computers

Contents

Note to Readers

This book introduces beginners to the principles of machine language: what it is, how it works, and how to program with it.

It is based on an intensive two-day course on machine language that has been presented many times over the past five years.

Readers of this book should have a computer on hand: students will learn by doing, not just by reading. Upon completing the tutorial material in this book, the reader will have a good idea of the fundamentals of machine language. There will be more to be learned; but by this time, students should understand how to adapt other material from books and magazines to their own particular computers.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The author and publisher of this book have used their best efforts in preparing this book and the programs contained in it. These efforts include the development, research, and testing of the programs to determine their effectiveness. The author and the publisher make no warranty of any kind, expressed or implied, with regard to these programs, the text, or the documentation contained in this book. The author and the publisher shall not be liable in any event for claims of incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of the text or the programs.

At time of publication, the Commodore 264 is still undergoing design changes. The name is being changed to the "PLUS/4", a related machine, the Commodore 16, has also been announced. Detailed design information is not available; but the information given in this book for the Commodore 264 should be generally accurate.

Note to Authors

Do you have a manuscript or a software program related to personal computers? Do you have an idea for developing such a project? If so, we would like to hear from you. The Brady Co. produces a complete range of books and applications software for the personal computer market. We invite you to write to David Culverwell, Publishing Director, Brady Communications Company Inc., Bowie, Maryland 20715.

Preface

This book is primarily tutorial in nature. It contains, however, extensive reference material, which the reader will want to continue to use.

No previous machine language experience is required. It is useful if the reader has had some background in programming in other languages, so that concepts such as loops and decisions are understood

Beginners will find that the material in this book moves at a fast pace. Stay with it; if necessary, skip ahead to the examples and then come back to reread a difficult area.

Readers with some machine language experience may find some of the material too easy; for example, they are probably quite familiar with hexadecimal notation and don't need to read that part. If this is the case, skip ahead. But do enter all the programming projects; if you have missed a point, you may spot it while doing an exercise.

Programming students learn by doing. The beginner needs to learn simple things about his or her machine in order to feel in control. The elements that are needed may be itemized as:

- Machine language. This is the objective, but you can't get there without the next two items.
- Machine architecture. All the machine language theory in the world will have little meaning unless the student knows such things as where a program may be placed in memory, how to print to the screen, or how to input from the keyboard.
- Machine language tools. The use of a simple machine language monitor to read and change memory is vital to the objective of making the computer do something in machine language Use of a simple assembler and elements of debugging are easy once you know them; but until you know them, it's hard to make the machine do anything.

Principles of sound coding are important. They are seldom discussed explicitly, but run as an undercurrent through the material. The objective is this: it's easy to do things the right way, and more difficult to do them the wrong way. By introducing examples of good coding practices early, the student will not be motivated to look for a harder (and inferior) way of coding.

It should be pointed out that this book deals primarily with machine language, not assembly language. Assembler programs are marvellous things. but they are

too advanced for the beginner. I prefer to see the student forming an idea of how the bytes of the program lie within memory. After this concept is firmly fixed in mind, he or she can then look to the greater power and flexibility offered by an assembler.

Acknowledgements

Introduction

Why learn machine language? There are three reasons. First, for speed; machine language programs are fast. Second, for versatility; all other languages are limited in some way, but not machine language. Third, for comprehension; since the computer really works in machine language only, the key to understanding how the machine operates is machine language.

Is it hard? Not really. It's finicky, but not difficult. Individual machine language instructions don't do much, so we need many of them to do a job. But each instruction is simple, and anyone can understand it if he or she has the patience.

Some programmers who started their careers in machine language find "higher level" languages such as BASIC quite difficult by comparison. To them, machine language instructions are simple and precise, whereas BASIC statements seem vague and poorly defined by comparison.

Where will this book take you? You will end up with a good understanding of what machine language is, and the principles of how to program in it. You won't be an expert, but you'll have a good start and will no longer be frightened by this seemingly mysterious language.

Will the skills you learn be transportable to other machines? Certainly. Once you understand the principles of programming, you'll be able to adapt. If you were to change to a non-Commodore machine that used the 6502 chip (such as Apple or Atari), you'd need to learn about the architecture of these machines and about their machine language monitors. They would be different, but the same principles would apply on all of them.

Even if you change to a computer that doesn't use a chip from the 6502 family, you will be able to adapt. As you pick through the instructions and bits of the Commodore machine, you will have learned about the principles of all binary computers. You will need to learn the new microprocessor's instruction set, but it will be much easier the second time around.

Do you need to be a BASIC expert before tackling machine language? Not at all. This book assumes you know a little about programming fundamentals: loops, branching, subroutines, and decision making. But you don't need to be an advanced programmer to learn machine language.
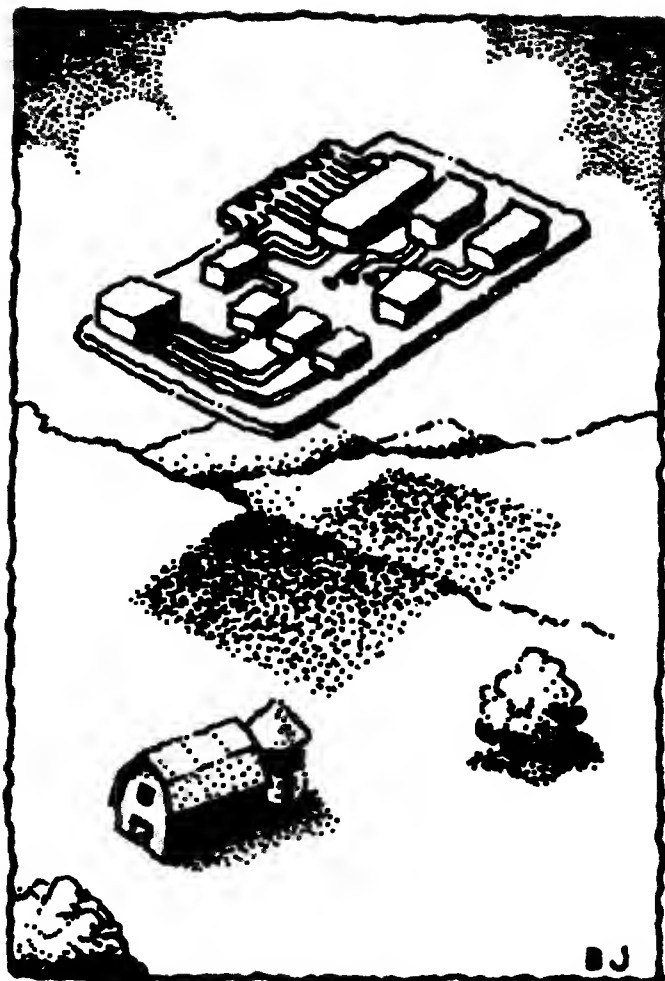
1

First
Concepts

This chapter discusses:

- The inner workings of microcomputers
- Computer notation: binary and hexadecimal
- The 650x's inner architecture
- Beginning use of a machine language monitor
- A computer's "memory layout"
- First machine language commands
- Writing and entering a simple program

The Inner Workings of Microcomputers

All computers contain a large number of electrical circuits. Within any binary computer, these circuits may be in only two states: "on" or "off."

Technicians will tell you that "on" usually means full voltage on the circuit concerned, and "off" means no voltage. There's no need for volume control adjustments within a digital computer: each circuit is either fully on or fully off.

The word "binary" means "based on two," and everything that happens within the computer is based on the two possibilities of each circuit: on or off. We can identify these two conditions in any of several ways:

<div align="center">

ON or OFF

TRUE or FALSE

YES or NO

1 or 0

</div>

The last description, 1 or 0, is quite useful. It is compact and numeric. If we had a group of eight circuits within the computer, some of which were "on" and others "off," we could describe their conditions with an expression such as:

11000111

This would signify that the two leftmost wires were on, the next three off, and the remaining three on. The value 11000111 looks like a number; in fact, it is a *binary* number in which each digit is 0 or 1. It should not be confused with the equivalent *decimal* value of slightly over 11 million; the digits would look the same, but in decimal each digit could have a value from 0 to 9. To avoid confusion with decimal numbers, binary numbers are often preceded by a percent sign, so that the number might be shown as %11000111.

Each digit of a binary number is called a *bit*, which is short for "binary digit." The number shown above has eight bits; a group of eight bits is a *byte*. Bits are often numbered from the right, starting at zero. The right-hand bit of the above number would be called "bit 0," and the left-hand bit would be called "bit 7." This may seem odd, but there's a good mathematical reason for using such a numbering scheme.

The Bus

It's fairly common for a group of circuits to be used together. The wires run from one microchip to another, and then on to the next. Where a group of wires are used together and connect to several different points, the group is called a *bus* (sometimes spelled "buss").

The PET, CBM, and VIC-20 use a microprocessor chip called the 6502. The Commodore 64 uses a 6510. The Commodore B series uses a 6509 chip, and the Commodore PLUS/4 uses a chip called 7501. All these chips are similar, and there are other chips in the same family with numbers like 6504; every one works on the same principles, and we'll refer to all of them by the family name 650x.

Let's take an example of a bus used on any 650x chip. A 650x chip has little built-in storage. To get an instruction or perform a computation, the 650x must call up information from "memory"—data stored within other chips.

The 650x sends out a "call" to all memory chips, asking for information. It does this by sending out voltages on a group of sixteen wires called the "address bus." Each of the sixteen wires may carry either voltage or no voltage; this combination of signals is called an *address*.

Every memory chip is connected to the address bus. Each chip reads the address, the combination of voltages sent by the processor. One and only one chip says, "That's me!" In other words, the specific address causes
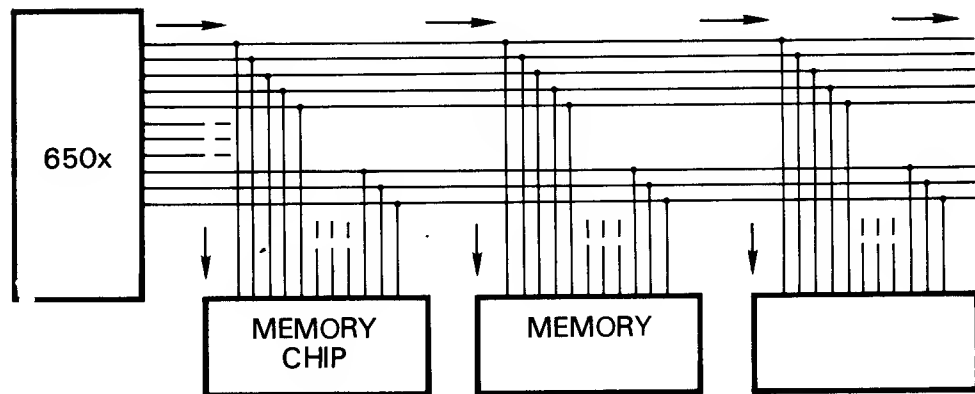
Figure 1.1 Address bus connecting 650x & 3 chips

that chip to be *selected*; it prepares to communicate with the 650x. All other chips say, "That's not me!" and will not participate in data transfer.

The Data Bus

Once the 650x microprocessor has sent an address over the address bus and it has been recognized by a memory chip, data may flow between memory and 650x. This data is eight bits (it flows over eight wires). It might look like this:

```
01011011
```

The data might flow either way. That is, the 650x might *read* from the memory chip, in which case the selected memory chip places information onto the data bus which is read by the microprocessor. Alternatively, the 650x might wish to *write* to the memory chip. In this case, the 650x places information onto the data bus, and the selected memory chip receives the data and stores it.
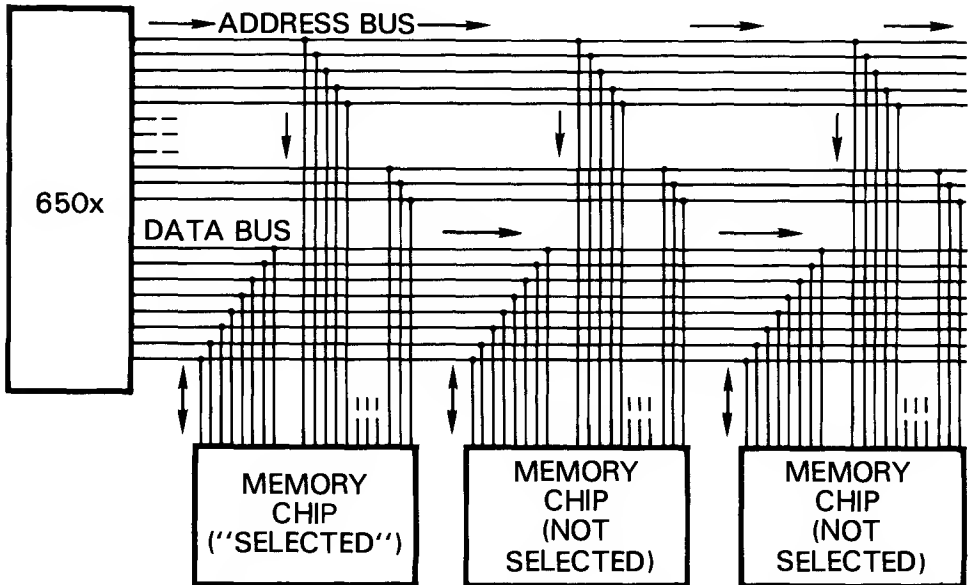
Figure 1.2: Two-way data bus

All other chips are still connected to the data bus, but they have not been selected, so they ignore the information.

The address bus is accompanied by a few extra wires (sometimes called

the *control bus*) that control such things as data timing and the direction in which the data should flow: read or write.

Number Ranges

The address bus has sixteen bits, each of which might be on or off. The possible combinations number 65536 (two raised to the sixteenth power). We then have 65536 different possibilities of voltages, or 65536 different addresses.

The data bus has eight bits, which allows for 256 possibilities of voltages. Each memory location can store only 256 distinct values.

It is often convenient to refer to an address as a decimal number. This is especially true for PEEK and POKE statements in the BASIC language. We may do this by giving each bit a "weight." Bit zero (at the right) has a weight of 1; each bit to the left has a weight of double the amount, so that bit 15 (at the left) has a weight of 32768. Thus, a binary address such as

    0001001010101100

has a value of 4096 + 512 + 128 + 32 + 8 + 4 or 4780. A POKE to 4780 decimal would use the above binary address to reach the correct part of memory.

128	64	32	16	8	4	2	1

EIGHT BITS

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

SIXTEEN BITS
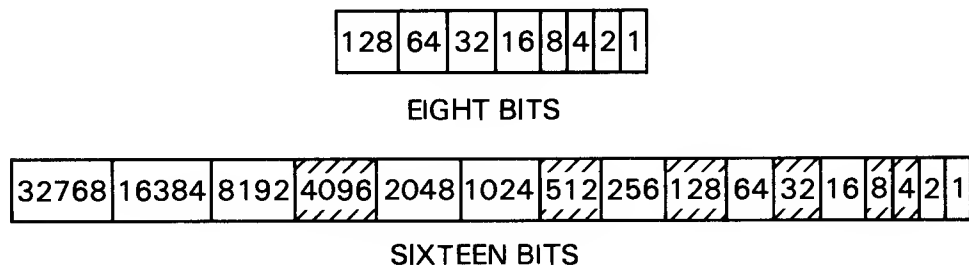
Figure 1.3

Direct conversion between decimal and binary is seldom needed. Such conversions usually pass through an intermediate number system, called hexadecimal.

Hexadecimal Notation

Binary is an excellent system for the computer, but it is inconvenient for most programmers. If one programmer asks another, "What address should

I use for some activity?", an answer such as "Address %0001001010101100" might be correct but would probably be unsatisfactory. There are too many digits.

Hexadecimal is a code used by humans to conveniently represent binary numbers. The computer uses binary, not hexadecimal; programmers use hexadecimal because binary is cumbersome.

To represent a binary number in hexadecimal, the bits must be grouped together four at a time. If we take the binary value given above and split it into groups of four, we get

    0001   0010   1010   1100

Now each group of four bits is represented by a digit as shown in the following table:

0000−0	0100−4	1000−8	1100−C
0001−1	0101−5	1001−9	1101−D
0010−2	0110−6	1010−A	1110−E
0011−3	0111−7	1011−B	1111−F

Thus, the number would be represented as hexadecimal 12AC. A dollar sign is often prefixed to a hexadecimal number so that it may be clearly recognized:$12AC.

The same type of weighting is applied to each bit of the group of four as was described before. In other words, the rightmost bit (bit zero) has a weight of 1, the next left a weight of 2, the next a weight of 4, and the leftmost bit (bit three) a weight of 8. If the total of the weighted bits exceeds nine, an alphabetic letter is used as a digit: A represents ten; B, eleven; C, twelve; and F, fifteen.

Eight-bit numbers are represented with two hexadecimal digits. Thus, %01011011 may be written as $5B.

Hexadecimal to Decimal

As we have seen, hexadecimal and binary numbers are easily interchangeable. Although we will usually write values in "hex," occasionally we will need to examine them in their true binary state to see a particular information bit.

Hexadecimal isn't hard to translate into decimal. You may recall that in early arithmetic we were taught that the number 24 meant, "two tens and four units." Similarly, hexadecimal 24 means "two sixteens and four units," or a decimal value of 36. By the way, it's better to say hex numbers as

"two four" rather than "twenty-four," to avoid confusion with decimal values.

The formal procedure, or *algorithm*, to go from hex to decimal is as follows.

Step 1. Take the leftmost digit; if it's a letter A to F, convert it to the appropriate numeric value (A equals 10, B equals 11, and so on)

Step 2· If there are no more digits, you're finished, you have the number. Stop.

Step 3: Multiply the value so far by sixteen. Add the next digit to the result, converting letters if needed Go back to step 2

Using the above steps, let's convert the hexadecimal number $12AC.

Step 1· The leftmost digit is 1.

Step 2: There are more digits, so we'll continue.

Step 3 1 times 16 is 16, plus 2 gives 18.

Step 2· More digits to come.

Step 3· 18 times 16 is 288, plus 10 (for A) gives 298

Step 2: More digits to come.

Step 3. 298 x 16 is 4768, plus 12 (for C) gives 4780.

Step 2: No more digits: 4780 is the decimal value

This is easy to do by hand or with a calculator.

Decimal to Hexadecimal

The most straightforward method to convert from decimal to hexadecimal is to divide repeatedly by 16; after each division, the remainder is the next hexadecimal digit, working from right to left. This method is not too well suited to small calculators, which usually don't give remainders. The following fraction table may offer some help:

.0000-0	.2500-4	.5000-8	.7500-C
.0625-1	.3125-5	.5625-9	.8125-D
.1250-2	.3750-6	.6250-A	.8750-E
.1875-3	.4375-7	.6875-B	.9375-F

If we were to translate 4780 using this method, we would divide by 16, giving 298.75. The fraction tells us the last digit is C; we now divide 298 by 16, giving 18.625. The fraction corresponds to A, making the last two digits AC. Next we divide 18 by 16, getting 1.125—now the last three digits are 2AC. We don't need to divide the one by 16, although that would work; we just put it on the front of the number to get an answer of $12AC.

There are other methods of performing decimal-to-hexadecimal conver-

sions. You may wish to look them up in a book on number systems. Alternatively, you may wish to buy a calculator that does the job electronically. Some programmers get so experienced that they can do conversions in their heads; I call them "hex nuts."

Do not get fixed on the idea of numbers. Memory locations can always be described as binary numbers, and thus may be converted to decimal or hexadecimal at will. But they may not *mean* anything numeric. the memory location may contain an ASCII coded character, an instruction, or any of several other things.

Memory Elements

There are generally three types of devices attached to the memory busses (address, data, and control busses):

- RAM. Random access memory. This is the read and write memory, where we will store the programs we write, along with values used by the program We may store information into RAM, and may recall the information at any time

- ROM. Read only memory. This is where the fixed routines are kept within the computer. We may not store information into ROM; its contents were fixed
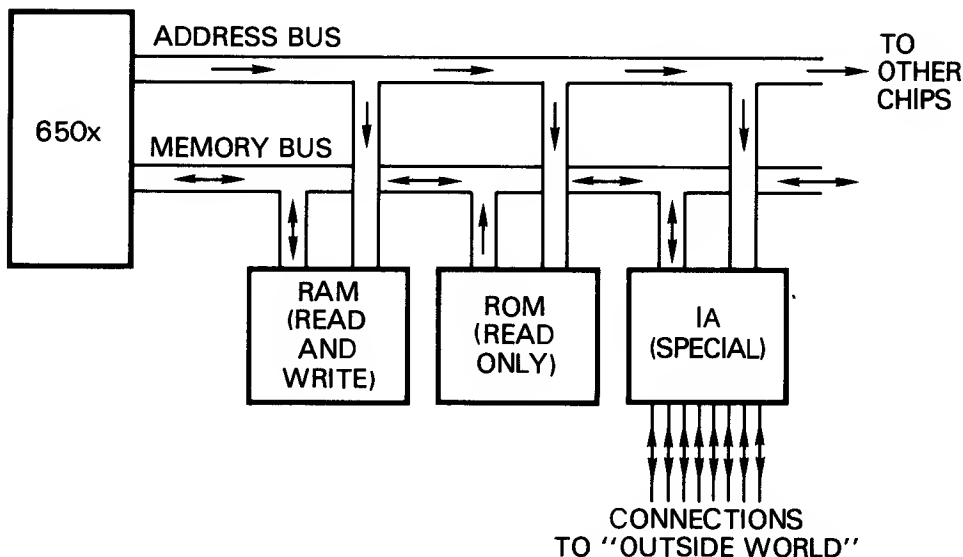
Figure 1.4

when the ROM was made We will use program units (subroutines) stored in ROM to do special tasks for us, such as input and output.

- IA. Interface adaptor chips These are not memory in the usual sense, but, these chips are assigned addresses on the address bus, so we call them "memory-mapped" devices Information may be passed to and from these devices, but the information is generally not stored in the conventional sense IA chips contain such functions as· input/output (I/O) interfaces that serve as connections to the "outside world", timing devices, interrupt control systems. and sometimes specialized functions, such as video control or sound generation. IA chips come in a wide variety of designs, including the PIA (peripheral interface adaptor), the VIA (versatile interface adaptor), the CIA (complex interface adaptor), the VIC (video interface chip), and the SID (sound interface device).

Within a given computer, some addresses may not be used at all. Some devices may respond to more than one address, so that they seem to be in two places in memory.

An address may be thought of as split in two parts. One part, usually the high part of the address, selects the specific chip. The other part of the address selects a particular part of memory within the chip. For example, in the Commodore 64. the hex address $D020 (decimal 53280) sets the border color of the video screen The first part of the address (roughly, $D0 ..) selects the video chip; the last part of the address (.. 20) selects the part of the chip that controls border color.

Microprocessor Registers

Within the 650x chip are several storage areas called *registers*. Even though they hold information, they are not considered "memory" since they don't have an address Six of the registers are important to us. Briefly, they are:

PC: (16 bits)	The program counter tells where the next instruction will come from.
A, X and Y (8 bits each)	These registers hold data.
SR	The status register, sometimes called PSW (processor status word), tells about the results of recent tests, data handling, and so on.
SP	The stack pointer keeps track of a temporary ' storage area.

We will talk about each of these registers in more detail later. At the moment, we will concentrate on the PC (program counter).

650x CHIP
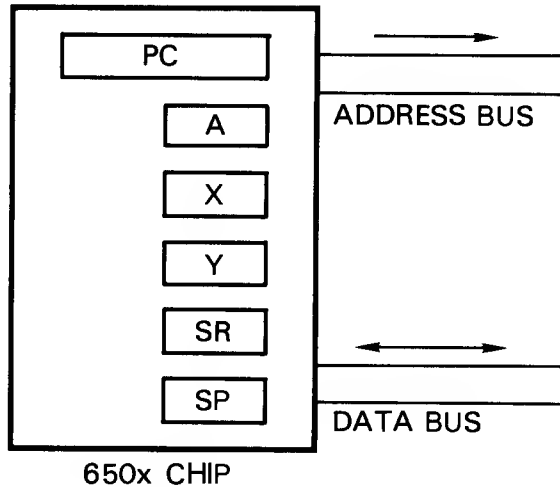
Figure 1.5

Instruction Execution

Suppose that the 650x is stopped (not an easy trick), and that there is a certain address, say $1234, in the PC. The moment we start the micro-computer, that address will be put out to the address bus as a read address, and the processor will add one to the value in the PC.

Thus, the contents of address $1234 will be called for, and the PC will change to $1235. Whatever information comes in on the data bus will be taken to be an *instruction*.

The microprocessor now has the instruction, which tells it to do something. The action is performed, and the whole action now repeats for the next
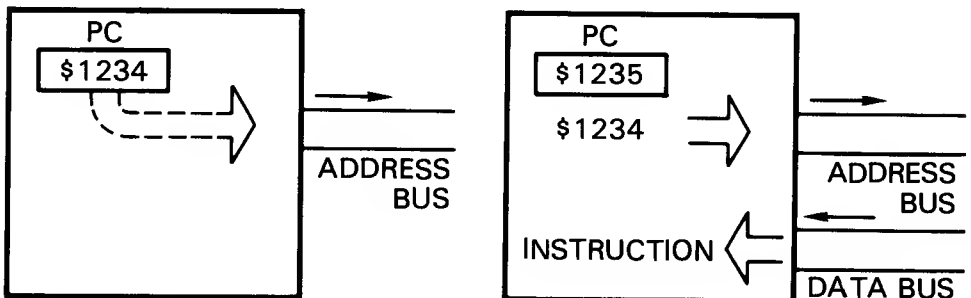
Figure 1.6 Arrow to addrs bus

instruction. In other words, address $1235 will be sent to memory, and the PC will be incremented to $1236.

You can see that the processor works in the same way that most computer languages do: an instruction is executed, and then the computer proceeds to the next instruction, and then the next, and so on. We can change the sequence of execution by means of a "jump" or "branch" to a new location, but normally, it's one instruction after another.

Data Registers: A, X, and Y

Any of three registers can be used to hold and manipulate eight bits of data. We may *load* information from memory into A, X, or Y; and we may *store* information into memory from any of A, X, or Y

Both "load" and "store" are copying actions If I load A (LDA) from address $2345, I make a copy of the contents of hex 2345 into A; but 2345 still contains its previous value. Similarly, if I store Y into $3456, I make a copy of the contents of Y into that address; Y does not change.

The 650x has no way of moving information directly from one memory address to another. Thus, this information must pass via A, X, or Y; we load it from the old address, and store it to the new address

Later, the three registers will take on individual identities. For example, the A register is sometimes called the accumulator, since we perform addition and subtraction there. For the moment, they are interchangeable: we may load to any of the three, and we may store from any of them.

First Program Project

Here's a programming task: locations $0380 and $0381 contain information. We wish to write a program to exchange the contents of the two locations. How can we do this?

We must make up a plan. We know that we cannot transfer information directly from memory to memory. We must load to a register, and then store. But there's more. We must not store and destroy data in memory until that data has been safely put away. How can we do this?

Here's our plan. We may load one value into A (say, the contents of $0380), and load the other value into X (the contents of $0381). Then we could store A and X back, the other way around.

We could have chosen a different pair of registers for our plan, of course:

A and Y, or X and Y. But let's stay with the original plan. We can code our plan in a more formal way:

```
LDA  $0380    (bring in first value)
LDX  $0381    (bring in second value)
STA  $0381    (store in opposite place)
STX  $0380    (and again)
```

You will notice that we have coded "load A" as LDA, "load X" as LDX, "store A" as STA, and "store X" as STX. Every command has a standard three-letter abbreviation called a *mnemonic*. Had we used the Y register, we might have needed to use LDY and STY.

One more command is needed. We must tell the computer to stop when it has finished the four instructions. In fact, we can't stop the computer; but if we use the command BRK (break), the computer will go to the *machine language monitor* (MLM) and wait for further instructions. We'll talk about the MLM in a few moments.

We have written our program in a notation styled for human readability, called *assembly language*. But the computer doesn't understand this notation. We must translate it to machine language.

The binary code for LDA is %10101101, or hexadecimal AD. That's what the computer recognizes; that's the instruction we must place in memory. So we code the first line.

```
AD 80 03        LDA   $0380
```

It's traditional to write the machine code on the left, and the source code on the right. Let's look closely at what has happened.

LDA has been translated into $AD. This is the *operation code*, or *op code*, which says what to do. It will occupy one byte of memory. But we need to follow the instruction with the address from which we want the load to take place. That's address $0380; it's sixteen bits long, and so it will take two bytes to hold the address. We place the address of the instruction, called the *operand*, in memory immediately behind the instruction. But there's a twist. The last byte comes first, so that address $0380 is stored as two bytes: 80 first and then 03.

This method of storing addresses—low byte first—is standard in the 650x. It seems unusual, but it's there for a good reason. That is, the computer gets extra speed from this "backwards" address. Get used to it; you'll see it again, many times

Here are some machine language op codes for the instructions we may use. You do not need to memorize them.

```
LDA-AD        LDX-AE        LDY-AC        BRK-00
STA-8D        STX-8E        STY-8C
```

Now we can complete the translation of our program.

```
AD 80 03        LDA $0380
AE 81 03        LDX $0381
8D 81 03        STA $0381
8E 80 03        STX $0380
00              BRK
```

On the right, we have our plan. On the left, we have the actual program that will be stored in the computer. We may call the right side *assembly code* and the left side *machine code*, to distinguish between them. Some users call the right-hand information *source code*, since that's where we start to plan the program, and the left-hand program *object code*, since that's the object of the exercise—to get code into the computer. The job of translating from source code to object code is called *assembly*. We performed this translation by looking up the op codes and translating by hand; this is called *hand assembly*

The code must be placed into the computer. It will consist of 13 bytes: AD 80 03 AE 81 03 8D 81 03 8E 80 03 00. That's the whole program. But we have a new question: where do we put it?

Choosing a Location

We must find a suitable location for our program. It must be placed into RAM memory, of course, but where?

For the moment, we'll place our program into the cassette buffer, starting at address $033C (decimal 828). That's a good place to put short test programs, which is what we will be writing for a while.

Now that we've made that decision, we face a new hurdle: how do we get the program in there? To do that, we need to use a machine language monitor.

Monitors: What They Are

All computers have a built-in set of programs called an *operating system* that gives the machine its style and basic capabilities. The operating sys-

tem takes care of communications—reading the keyboard, making the proper things appear on the screen, and transferring data between the computer and other devices, such as disk, tape, or printer.

When we type on the computer keyboard, we use the operating system, which detects the characters we type. But there's an extra set of programs built into the computer that must decide what we *mean*. When we are using the BASIC language, we'll be communicating with the *BASIC monitor*, which understands BASIC commands such as NEW, LOAD, LIST, or RUN. It contains *editing* features that allow us to change the BASIC program that we are writing.

But when we switch to another system—often another language—we'll need to use a different monitor. Commands such as NEW or LIST don't have any meaning for a machine language program. We must leave the BASIC monitor and enter a new environment: the *machine language monitor*. We'll need to learn some new commands because we will be communicating with the computer in a different way.

The Machine Language Monitor

Most PET/CBM computers have a simple MLM (machine language monitor) built in. It may be extended with extra commands. The Commodore PLUS/4 contains a very powerful MLM. The VIC-20 and Commodore 64 do not have a built-in MLM, but one can be added. Such a monitor may be either loaded into RAM or plugged in as a cartridge. Monitors may be purchased or obtained from user clubs.

Most machine language monitors work in a similar way, and have about the same commands. To proceed, you'll need an MLM in your computer. Use the built-in one, plug it in, load it in, or load and run . . . whatever the instructions tell you. On a PET/CBM machine, typing the command SYS 4 will usually switch you to the built-in monitor. After an MLM has been added to a VIC or Commodore 64, the command SYS 8 will usually get you there. On the Commodore PLUS/4, the BASIC command MONITOR will bring the monitor into play.

Monitor Display

The moment you enter the MLM, you'll see a display that looks something like this:

```
B*
     PC   SR  AC  XR  YR  SP
.;  0005  20  54  23  6A  F8
.
```

The cursor will be flashing to the right of the period on the bottom line. The exact appearance of the screen information may vary according to the particular monitor you are using. Other material may be displayed—in particular, a value called IRQ—which we will ignore for the time being.

The information you see may be interpreted as follows:

B*—we have reached the MLM by means of a "break." More about that later

PC—The value shown below this title is the contents of the program counter. This indicates where the program "stopped." In other words, if the value shown is address 0005, the program stopped at address 0004, since the PC is ready to continue at the following address. The exact value (0004 versus 0005) may vary depending on the particular MLM.

SR—The value shown below shows the status register, which tells us the results of recent tests and data operations We'd need to split apart the eight bits and look at them individually to establish all the information here, we will do this at a later time.

AC, XR, and YR—The values shown below these three titles are the contents of our three data registers: A, X, and Y

SP—The value shown below is that of the stack pointer, which indicates a temporary storage area that the program might use. A value of F8, for example, tells us that the next item to be dropped into the stack area would go to address $01F8 in memory More on this later.

The period is roughly the equivalent of the READY statement in BASIC. It indicates that the computer is ready to receive a command from you.

You will notice that the display printed by the monitor (called the register display) shows the internal registers within the 650x chip. Sometimes there is another item of information, titled IRQ, in this display. It doesn't belong, since it does not represent a microprocessor register. IRQ tells us to what address the computer will go if an *interrupt* occurs; this information is stored in memory, not within the 650x.

MLM Commands

The machine language monitor is now waiting for you to enter a command. The old BASIC commands don't work any more; LIST or NEW or SYS are not known to the MLM. We'll list some popular commands in a moment. First, let's discuss the command that takes us back to BASIC.

. X exits the MLM and returns to the BASIC monitor. Try it. Remember to press RETURN after you've typed the X, of course. You will return to the BASIC system, and the BASIC monitor will type READY. You're back in familiar territory. Now go back to the monitor with SYS4 or SYS8 or MONITOR as the case may be. BASIC ignores spaces: it doesn't matter if you type SYS8 or SYS 8; just use the right number for your machine (4 for PET/CBM. 8 for VIC/64).

Remember: BASIC commands are no good in the MLM, and machine language monitor commands (such as . X) are no good in BASIC. At first, you'll give the wrong commands at the wrong time because it's hard to keep track of which monitor system is active If you type in an MLM command when you're in BASIC, you'll probably get a ?SYNTAX ERROR reply. If you type in a BASIC command when you're in the machine language monitor, you'll probably get a question mark in the line you typed.

Some other MLM commands are as follows (the prompting period is included):

.M 1000 1010   (display memory from hex 1000 to
                                        1010)
.R   (display registers ... again!)
.G 033C   (go to 033C and start running a pro-
                                        gram)

Do not enter this last (. G) command. There is no program at address 033C yet, so the computer would execute random instructions and we would lose control.

There are two other fundamental instructions that we won't use yet: they are . S for save and . L for load. These are tricky. Until you learn about BASIC pointers (Chapter 6), leave them alone.

Displaying Memory Contents

You'll notice that there is a command for displaying the contents of memory, but there doesn't seem to be one for changing memory. You can do both, of course.

Suppose we ask to display memory from $1000 to $1010 with the command

.M 1000 1010

Be careful that you have exactly one space before each address. You might get a display that looks something like this:

```
.:1000 11 3A E4 00 21 32 04 AA
.:1008 20 4A 49 4D 20 42 55 54
.:1010 54 45 52 46 49 45 4C 44
```

The four-digit number at the start of each line represents the address in memory being displayed. The two-digit numbers to the right represent the contents of memory. Keep in mind that all numbers used by the machine language monitor are hexadecimal.

In the example above, $1000 contains a value of $11; $1001 contains a value of $3A; and so on, until $1007, which contains a value of $AA. We continue with address $1008 on the next line. Most monitors show eight memory locations on each line, although some VIC-20 monitors show only five because of the narrow screen.

We asked for memory locations up to address $1010 only; but we get the contents of locations up to $1017 in this case The monitor always fills out a line, even if you don't ask for the extra values.

Changing Memory Contents

Once we have displayed the contents of part of memory, we can change that part of memory easily. All we need to do is to move the cursor until it is positioned over the memory contents in question, type over the value displayed, and then press RETURN

This is quite similar to the way BASIC programs may be changed; you may type over on the screen, and when you press RETURN, the new line replaces the old. The general technique is called *screen editing*.

If you have displayed the contents of memory, as in the example above, you might like to change a number of locations to zero. Don't forget to strike RETURN so that the change on the screen will take effect in memory. Give another .M memory display command to confirm that memory has indeed been changed.

Changing Registers

We may also change the contents of registers by typing over and pressing RETURN. You may take a register display with command .R, and then change the contents of PC, AC, XR, and YR. Leave the contents of SR and SP unchanged—tricky things could happen unexpectedly if you experiment with these two.

Entering the Program

We might rewrite our program one last time, marking in the addresses that each instruction will occupy. You will recall that we have decided to put our program into memory starting at address $033C (part of the cassette buffer).

```
033C  AD  80  03      LDA  $0380
033F  AE  81  03      LDX  $0381
0342  8D  81  03      STA  $0381
0345  8E  80  03      STX  $0380
0348  00
```

Remember that most of the above listing is cosmetic. The business end of the program is the set of two-digit hex numbers shown to the left. At the extreme left, we have addresses—that's information, but not the program. At the right, we have the "source code"—our notes on what the program means.

How do we put it in? Easy. We must change memory. So, we go to the MLM, and display memory with

.M 033C 0348

We might have anything in that part of memory, but we'll get a display that looks something like

```
.:033C  xx  xx  xx  xx  xx  xx  xx  xx
.:0344  xx  xx  xx  xx  xx  xx  xx  xx
```

You won't see "xx," of course; there will be some hexadecimal value printed for each location. Let's move the cursor back and change this display so that it looks like this:

```
.:033C  AD  80  03  AE  81  03  8D  81
.:0344  03  8E  80  03  00  xx  xx  xx
```

Don't type in the "xx"—just leave whatever was there before. And be sure to press RETURN to activate each line; if you move the cursor down to get to the next line without pressing RETURN, the memory change would not happen.

Display memory again (.M   033C   0348) and make sure that the program is in place correctly. Check the memory display against the program listing, and be sure you understand how the program is being transcribed into memory.

If everything looks in order, you're ready to run your first machine language program.

Preparation

There's one more thing that we need to do. If we want to swap the contents of addresses $0380 and $0381, we'd better put something into those two locations so that we'll know that the swap has taken place correctly.

Display memory with . M   0380   0381 and set the resulting display so that the values are

.:0380 11 99 xx xx xx xx xx xx

Remember to press RETURN. Now we may run our program; we start it up with

.G 033C

The program runs so quickly that it seems instantaneous (the run time is less than one fifty thousandth of a second). The last instruction in our program was BRK for break, and that sends us straight to the MLM with a display of *B (for break, of course) plus all the registers.

Nothing seems to have changed. But wait. Look carefully at the register display. Can you explain the values you see in the AC and XR registers? Can you explain the PC value?

Now you may display the data values we planned to exchange. Give the memory display command . M   0380   0381—have the contents of the two locations changed?

They'd better have changed Because that's what the writing of our program was all about

Things You Have Learned

—Computers use binary If we want to work with the inner fabric of the computer, we must come to terms with binary values.

—Hexadecimal notation is for humans, not for computers. It's a less clumsy way for people to cope with binary numbers.

—The 650x microprocessor chip communicates with memory by sending an address over its memory bus.

—The 650x has internal work areas called registers.

—The program counter tells us the address from which the processor will get its next instruction.

—Three registers, called A, X, and Y, are used to hold and manipulate data. They may be loaded from memory, and stored into memory.

—Addresses used in 650x instructions are "flipped": the low byte comes first, followed by the high byte.

—The machine language monitor gives us a new type of communications path into the computer. Among other things, it allows us to inspect and change memory in hexadecimal.

Detail: Program Execution

When we say .G 033C to start up our program, the microprocessor goes through the following steps:

1 It asks for the contents of $033C; it receives $AD, which it recognizes as the op code "load A." It realizes that it will need a two-byte address to go with this instruction

2 It asks for the contents of $033D, and then $033E. As it receives the values of $80 and $03 it gathers them into an "instruction address."

3 The microprocessor now has the whole instruction The PC has moved along to $033F The 650x now *executes* the instruction It sends address $0380 to the address bus; when it gets the contents (perhaps $11), it delivers this to the A register The A register now contains $11.

4. The 650x is ready to take on the next instruction, the address $033F goes from the PC out to the address bus; and the program continues.

Questions and Projects

Do you know that your computer has a part of memory called "screen memory"? Whatever you put into that part of memory appears on the screen. You'll find this described in BASIC texts as "screen POKE-ing."

The screen on the PET/CBM is at $8000 and up; on the VIC, it's often (but not always) at $1E00 and up; on the Commodore 64, it's usually at $0400; and on the PLUS/4, it may be found at $0C00.

If you write a program to store information in the screen memory address, the appropriate characters will appear on the screen. You might like to try this. You can even "swap" characters around on the screen, if you wish.

Two pitfalls may arise. First, you might write a perfect program that places information near the top of the screen; then, when the program finishes, the screen might scroll, and the results would disappear. Second, the VIC and Commodore 64 use color, and you might inadvertently produce white-on-white characters; these are hard to see.

Here's another question. Suppose I asked you to write a program to move the contents of five locations, $0380 to $0384, in an "end-around" fashion, so that the contents of $0380 moved to $0381, $0381 to $0382, and so on, with the contents of $0384 moved to $0380. At first glance, we seem to have a problem: we don't have five data registers, we have only three (A, X, and Y). Can you think of a way of doing the job?
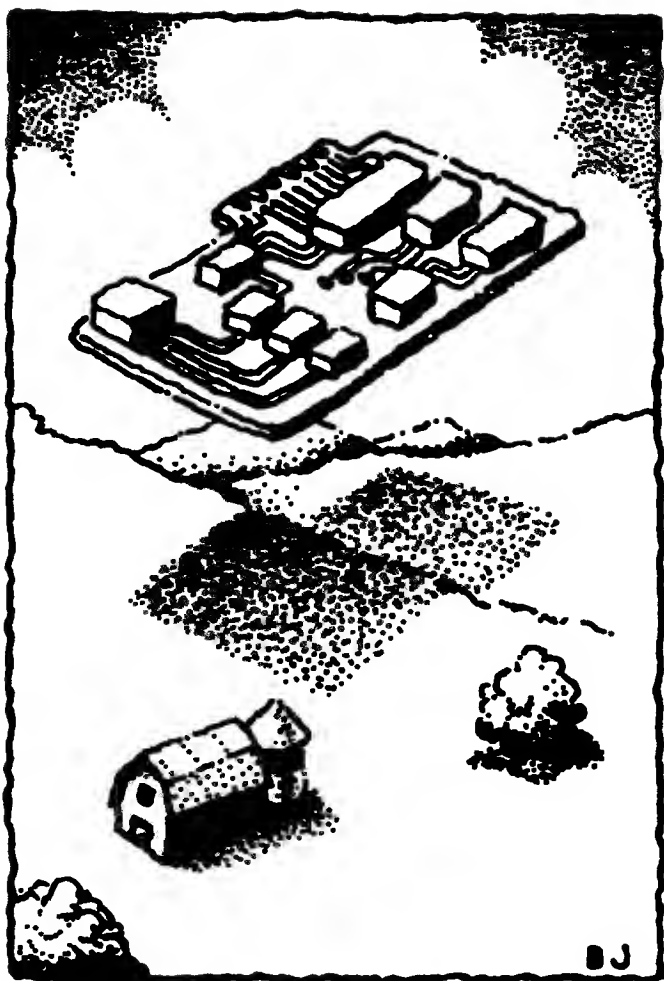
2

Controlling Output

This chapter discusses:

- Calling machine language subroutines
- The PRINT subroutine
- Immediate addressing
- Calling machine language from BASIC
- Tiny assembler programs
- Indexed addressing
- Simple loops
- Disassembly

Calling Machine Language Subroutines

In BASIC, a "package" of program statements called a *subroutine* may be brought into action with a GOSUB command. The subroutine ends with a RETURN statement, which causes the program to return to the *calling point*, i.e., the statement immediately following GOSUB.

The same mechanism is available in machine language. A group of instructions may be invoked with a *jump subroutine* (JSR) command. The 650x goes to the specified address and performs the instructions given there until it encounters a *return from subroutine* (RTS) command, at which time it resumes execution of instructions at the calling point: the instruction immediately following JSR.

For example, if at address $033C I code the instruction JSR $1234, the 650x will change its PC to $1234 and start to take instructions from that address. Execution will continue until the instruction RTS is encountered. At this time, the microprocessor would switch back to the instruction following the JSR, which in this case would be address $033F (the JSR instruction is three bytes long).

As in BASIC, subroutines may be "nested;" that is, one subroutine may call another, and that subroutine may call yet another. We will deal with subroutine mechanisms in more detail later. For the moment, we'll concern ourselves with calling prewritten subroutines.

Prewritten Subroutines

A number of useful subroutines are permanently stored in the ROM memory of the computer. All Commodore machines have a standard set of subroutines that may be called up by your programs. They are always at the same addresses, and perform in about the same way regardless of which Commodore machine is used: PET, CBM, Commodore 64, PLUS/4, or VIC-20. These routines are called the *kernal* subroutines. Details on them can be found in the appropriate Commodore reference manuals, but we'll give usage information here.

The original meaning of the term *kernal* seems to be lost in legend. It was originally an acronym, standing for something like "Keyboard Entry Read, Network and Link." Today, it's just the label we apply to the operating system that makes screen, keyboard, other input/output and control mechanisms work together. To describe this central control system, we might choose to correct the spelling so as to get the English word, "kernel." For now, we'll use Commodore's word.

The three major kernal subroutines that we will deal with in the next few chapters are shown here:

Address	Name	What it does
$FFD2	CHROUT	Outputs an ASCII character
$FFE4	GETIN	Gets an ASCII character
$FFE1	STOP	Checks the RUN/STOP key

With the first two subroutines, we can input and output data easily. The third allows us to honor the RUN/STOP key, to guard against certain types of programming error. In this chapter, we'll use CHROUT to print information to the screen.

CHROUT—The Output Subroutine

The CHROUT subroutine at address $FFD2 may be used for all types of output: to screen, to disk, to cassette tape, or to other devices. It's similar to PRINT and PRINT#, except that it sends only one character. For the moment, we'll use CHROUT only for sending information to the computer screen.

Subroutine·   CHROUT
Address. $FFD
Action.   Sends a copy of the character in the A register to the output channel. The output channel is the computer screen unless arrangements have been made to switch it

The character sent is usually ASCII (or PET ASCII) When sent to the screen, all special characters—graphics, color codes, cursor movements—will be honored in the usual way.

Registers: All data registers are preserved during a CHROUT call. Upon return from the subroutine, A, X, and Y will not have changed.

Status: Status flags may be changed. In the VIC and Commodore 64, the C (carry) flag indicates some type of problem with output

To print a letter X on the screen, we would need to follow these steps:

1 Bring the ASCII letter X ($58) into the A register,

2. JSR to address $FFD2.

Why Not POKE?

It may seem that there's an easier way to make things appear on the screen. We might POKE information directly to *screen memory*; in machine language, we would call this a *store* rather than a POKE, of course. The moment we change something in this memory area, the information displayed on the screen will change. Screen memory is generally located at the following addresses:

PET/CBM· $8000   and up (decimal 32768)

Commodore 64: $0400   and up (decimal 1024)

264/364 $0C00   and up (decimal 3072)

VIC-20. $1E00   and up (decimal 7680)

The screen memory of the VIC-20 in particular may move around a good deal, depending on how much additional RAM memory has been fitted.

Occasionally, screen POKEs are the best way to do the job. But most of the time we'll use the CHROUT, $FFD2 subroutine. Here are some of the reasons why:

- As with PRINT, we won't need to worry about where to place the next character, it will be positioned automatically at the cursor point.
- If the screen is filled, scrolling will take place automatically.
- Screen memory needs special characters. For example, the character X has a standard ASCII code of $58, but to POKE it to the screen we'd need to use the code $18. The CHROUT subroutine uses $58.   `
- Screen memory may move around, depending on the system and the program. The POKE address would need to change; but CHROUT keeps working
- Special control characters are honored: $0D for RETURN, to start a new line; cursor movements; color changes. We can even clear the screen by loading the screen-clear character ($93) and calling $FFD2.
- To POKE the screen of the VIC or Commodore 64, the corresponding color nybble memory must also be POKEd (see the appropriate memory map in Appendix B). With the subroutine at $FFD2, color is set automatically.

A Print Project

Let's write some code to print the letter H on the screen. Once again, we'll use address $033C, the cassette buffer, to hold our program. Reminder: be sure to have your monitor loaded and ready before you start this project.

First, the plan; we lay out the instructions

```
LDA #$48
```

We're using a new symbol (#) to signal a special type of information. It goes by a variety of names: pounds sign, sharp, hash mark, or numbers sign. A more formal name for the symbol is *octothorpe*, meaning "eight points." Whatever you call it, the symbol means "the following information is not an address, it's a value." In other words, we don't want the computer to go to address $48, we want it to load the A register with the *value* $48, which represents the ASCII letter H. This type of information access is called *immediate* addressing. In other words, take the information immediately, don't go to memory for it.

```
JSR $FFD2
```

The previous instruction brought the letter H into the A register; this one prints it to the screen. Now all we need to do is quit. BRK takes us to the machine language monitor.

Monitor Extensions

We could repeat the steps of the previous chapter: hand-assembling the source code into machine language, and then placing it into memory. We would need to know the instruction codes, and then do a careful translation. But there's an easier way.

Most machine language monitors contain extra commands to help us do this type of mechanical translation. We'll use the *assembler* feature of these monitors.

Most monitors contain the assemble (. A) command. The notable exception is the built-in monitors within the PET/CBM; these, however, can be extended by loading in a "monitor extension" program such as Supermon. The Commodore PLUS/4 series contains an extended monitor, which includes the . A command.

These assemblers are often called *nonsymbolic* assemblers. This means that whenever an address is needed, you must furnish that exact address. You cannot type in a name such as CHROUT and expect the tiny assembler to know what address that represents; instead, you must type $FFD2.

Load your monitor or monitor extension. Do any setup that may be needed. Then type the following monitor command:

```
.A 033C  LDA  #$48
```

We are asking the computer to assemble (. A) at address $033C (note we don't use the $ here) the command LDA, Load A, the immediate value of $48, which represents the ASCII letter H. When you press RETURN after entering this line, the computer may do either of two things:

1. It may do nothing except print a question mark somewhere on the line. The question mark indicates an error in your coding If the question mark appears directly after the letter A, your monitor does not understand the . A assemble instruction; get another monitor or properly set up the one you have.

2. Or, it will correctly translate your instruction, and put the object code into memory starting at the address specified. In this case, that would happen to be $A9 at address $033C and $48 at address $033D. It would then help you by printing part of the next expected instruction The computer expects that you will type a line starting with

    . A 033E

It places the first part of this line on the screen to save you typing. The screen should now look like this:

```
. A 033C   LDA   #$48
. A 033E
```

You may now complete the instruction by typing in JSR $FFD2 and pressing RETURN. Again, the computer will anticipate your next line by printing . A 0341, which allows you to type in the final command, BRK. The screen now looks like this:

```
. A 033C   LDA   #$48
. A 033E   JSR   $FFD2
. A 0341   BRK
. A 0342
```

The computer is still waiting for another instruction. We have no more instructions, so we press RETURN to signal that we're finished.

At this point, our program is stored in memory. The instructions have been *assembled* directly into place, and the object code is hopefully ready to go

Note that this saves us the trouble of remembering—or looking up—the op codes for each instruction. And we don't need to keep track of how long each instruction should be; the assembler does it for us.

If you like, you can display memory and look at the object program with the . M   033C   0341. You'll see the bytes of your program in memory:

```
. :033C A9 48 20 D2 FF 00 xx xx
```

The first six bytes are your program. The last two bytes don't matter: they were whatever was in that part of memory before. We don't care what is there, since the program will stop when it reaches the BRK ($00) at address $0341; it won't be concerned with the contents of memory at $0342 or $0343.

Checking: The Disassembler

When we changed our source code into object code, we called this process of translation *assembly*, and we called a program that did the job an *assembler*.

Now we've written a program and it's safely stored in memory. We have inspected memory and have seen the bytes there; but they are hard to read. It would be convenient if we could perform an inverse assembly, that is, take the contents of memory and translate it into source code. The monitor has this capability, called a *disassembler*.

If we ask the computer to *disassemble* the code starting at $033C, it will examine the code there and establish that the contents ($A9) correspond to an LDA immediate command. It will then print for our information LDA #$48, which is much more readable than the original two bytes, A9   48.

Give the command . D 033C and press RETURN. D stands for disassemble. of course. and the address must follow.

The computer will now show a full screen of code. On the left is the address followed by the bytes making up the instruction. On the right is the reconstructed source code. The screen shows much more memory than our program needs. Again, we ignore all lines beyond address $0341, which is the last instruction of our program. Anything following is "junk" left in memory that the program does not use.

An interesting feature of most disassembly listings is that the cursor is left flashing on the last line of the disassembly rather than on the line below. When you have a large program, this allows you to type the letter D followed by RETURN and the next part of your program will immediately be displayed. On the other hand, if you don't want to disassemble more code, press the cursor down key and move to a "clean" line before typing your next instruction.

A disassembly is a good way to check for errors. If you find an error in the listing, you may correct that line by re-assembling it, using the . A command once again. Minor errors may be corrected directly on the left-hand side of the disassembly listing. In other words, suppose that you had

incorrectly coded LDA #$5ß during the assembly phase; when you perform the disassembly, this line will show as

., 033C A9 58   LDA #$58

You recognize that the 58 should be 48; you may move the cursor up—use cursor home if you wish—and type over the value *on the left-hand side*. In this case, you place the cursor over the 5, type 4 to change the display to 48, and press RETURN. You will see from the display that the problem has been fixed.

Running the Program

If necessary, move the cursor down to an empty line. Type the command .G 033C and the program will run. Again, it doesn't take long; the break back to the MLM seems instantaneous. Where's the letter H that we were supposed to print? It's hard to see, but it's there. Look at your .G 033C command and you'll see it.

Project for enthusiasts: Can you add to the program and print HI? The ASCII code for the letter I is $49. Can you add again and print HI on a separate line? The ASCII code for a RETURN is $0D. Remember that you can find all ASCII codes in Appendix D; look in the column marked ASCII.

Linking with BASIC

So far we have started up our programs with a .G (go) command from the MLM, and we have terminated our programs with a BRK command that returns us to the monitor. That's not a convenient way to run a program; most users would prefer to say RUN out of BASIC and have the computer do everything.

We can link to a machine language program from BASIC and when the program is finished, it can return to BASIC and allow the BASIC program to continue to run. The commands we need are

(BASIC) SYS—Go to a machine language subroutine at the stated address.

(Machine language) RTS—Return to whoever called this subroutine

Let's change our machine language program first. We must change the BRK at the end to RTS (return from subroutine) so that when the program is finished it will return to BASIC. If you like, you may change it directly on the disassembly listing: disassemble and then type over the 00 byte

that represents BRK with a value of 60. Press RETURN and you'll see that the instruction has now changed to RTS. Alternatively, you may re-assemble with

```
.A 033C LDA #$48
.A 033E JSR $FFD2
.A 0341 RTS
```

Now return to BASIC (using the .X command). The computer will say READY; you may now call your program with a SYS command.

Address $033C is 828 in decimal. Thus, we type SYS 828. When we press RETURN, the letter H will be printed.

We're not finished. Any machine language subroutine may be called from a BASIC program. Type NEW, which clears out the BASIC work area; our machine language program is left untouched, since NEW is a BASIC command. Now enter the following program:

```
100 FOR   J = 1 TO·10
110 SYS   828
120 NEXT   J
```

How many times will our program at 828 ($033C) be called? How many times will the letter H be printed? Will they be on the same line or separate lines? Type RUN and see.

Project for enthusiasts: Again, change the machine language program to say HI. Use your imagination. What else would you like the computer to say? Would you like to use colors or reverse font?

We've achieved an important new plateau: BASIC and machine language working together. It's easier on the user, who doesn't have to learn specialized monitor commands. It's easier on the programmer, too, since things that are easy to do in BASIC can be written in that language; things that are clumsy or slow in BASIC can be written in machine language. We can get the best of both worlds.

Let's distinguish our three different types of subroutine calls:

GOSUB—calls a BASIC subroutine from a BASIC program.

SYS—calls a machine language subroutine from a BASIC program

JSR—calls a machine language subroutine from machine language.

Loops

We know how to send characters to the screen, one at a time But long messages, such as THE QUICK BROWN CAT . . ., might lead to te-

dious coding if we had to write an instruction for each letter to be sent. We need to set up a program loop to repeat the printing activity.

Let's write a program to print the word HELLO followed by a RETURN.

We must store the word HELLO somewhere in memory. It doesn't matter where, provided it doesn't conflict with anything else. I'll arbitrarily choose address $034A to $034F. We'll put it there in a moment. Remember that the characters that make up the word HELLO (plus the RETURN) are not program instructions; they are simple data. We must put them in place with a memory change—we *must not try to assemble them*.

We will need to count the characters as we send them. We wish to send six characters, so a count of six is our limit. Let's use the X register to keep track of the count. First, we must set X to zero:

`.A 033C LDX #$00`

Note that we use the # symbol to denote an immediate value: we want to load X with the value zero, not something from address 0. Now, we'll do something new. I want to take a character to be printed from address $034A. But wait, that's only the first time around. When we come back to this point in the loop, I want to take a character from $034B, and then from $034C, and so on.

How can we do this? It seems that we must write one address into the LDA instruction, and that address can't change. But there is a way.

We can ask the computer to take the address we supply, and add the contents of X or Y to this address before we use it. The computed address is called an *effective address*.

Let's look at our position. The first time around the loop, X is counting the characters and has a value of zero. If we specify our address as 034A + X, the effective address will be 034A. That's where we will have stored the letter H.

When we come back around the loop—we haven't written that part yet— X should now equal one. An address of 034A + X would give an effective address of 034B; the computer would go there and get the letter E. As we go around the loop, the letters, L, L, O, and RETURN will be brought in as needed.

As we enter the LDA instruction, we don't type the plus sign. Instead, we signal indexing with a comma: LDA $034A,X. We may use either X or Y for indexing: they are sometimes called *index registers*. In this case, of course. we use X. So we code

```
.A 033E    LDA $034A,X
.A 0341    JSR $FFD2
```

The first time, the computer loads the contents of address $034A (the letter H of HELLO) and prints it. When the loop comes back here, with X equal to one, this instruction will load the contents of $034B and print the letter E.

The X register counts the number of letters printed, so we must add one to the contents of X. There's a special command that will add one to the contents of X: INX, for increment X. A similar code, INY, allows Y to be incremented; and DEX (decrement X) and DEY (decrement Y) allow X or Y to be decremented, or reduced, by one. At the moment, INX is the one we need for counting:

```
.A 0344    INX
```

Now we can test X to see if it is equal to six yet. The first time around, it won't be since X started at zero and was incremented to a value of 1. If X is not equal to six, we'll go back to $033E and print another letter. Here's how we code it:

```
.A 0345    CPX #$06
.A 0347    BNE $033E
```

CPX stands for compare X; note that we are testing for an immediate value of six, so we use the # symbol. BNE means branch not equal: if X is not equal to six, back we go to address $033E.

A little careful thought will reveal that the program will go back five times for a total of six times around the loop. It's exactly what we want.

Let's show the whole code, completing it with RTS:

```
.A 033C    LDX #$00
.A 033E    LDA $034A,X
.A 0341    JSR $FFD2
.A 0344    INX
.A 0345    CPX #$06
.A 0347    BNE $033E
.A 0349    RTS
```

We may now put the characters for HELLO into memory. These are data, not instructions, so we must not try to assemble them. Instead, we change memory in the usual way, by displaying and then typing over. We give the command .M 034A 034F, and type over the display to show

```
.:034A    48 45 4C 4C 4F 0D xx xx
```

By a lucky coincidence, this data fits exactly behind our program.

Everything should be ready now Disassemble the program at $033C and check it. You may note that the data at $034A doesn't disassemble too well, but that's to be expected; these bytes are not instructions and cannot be decoded.

When all looks well, return to BASIC (with .X) and try SYS 828. The computer should say HELLO.

Once again, set up a BASIC loop program:

```
100 FOR  J=1 TO 3
110 SYS   828
120 NEXT   J
```

A Comment on SAVE

If you wished to save the program to cassette tape, you'd have a problem on the VIC or Commodore 64. The machine language program is in the cassette buffer; a save-to-tape command would cause the contents of that buffer to be destroyed before the program could be written to tape. Even disk commands would not be completely safe: 4.0 BASIC disk commands use the cassette buffer area as a work area; using these commands would probably destroy our machine language program.

But saving the program is not the main problem. A correctly saved program can give trouble when you try to bring it back and run it safely. The difficulty is related to BASIC pointers, especially the start-of-variables pointer. The problem, and how to solve it, will be discussed in some detail in Chapter 6

A Stopgap SAVE

We can preserve short programs by making them part of DATA statements. The procedure is not difficult if screen editing is used intelligently.

We note that the program extends from $033C to $034F, including the message (HELLO) at the end. The decimal equivalents to these addresses are 828 to 847. Enter the following BASIC line:

```
FOR J=828 TO 847:PRINT PEEK(J);:NEXT J
```

Study the above line. You will see that it asks BASIC to go through the part of memory containing your machine language program, and display the contents (in decimal notation, of course). You'll see a result that looks something like this:

```
162  0  189  74  3  32  210  255  232  224  6  208  245  96
72  69  76  76  79  13
```

These are indeed the bytes that make up your program. With a little study, you could reconstruct the 162-0 combination to be LDX #$00, or the 72-69-76-76-79 at the end to be the word HELLO in ASCII. It looks different when it's in decimal, but it's still the same numbers.

You may try a little skill and artistry, using screen editing to perform the next activity, or you may just retype the numbers into data lines a shown. Either way, arrange the numbers as follows:

```
50 DATA 162,0,189,74,3,32,210,255,232,224,6
60 DATA 208,245,96,72,69,76,76,79,13
```

We now have a copy of our program, exactly the way it appears in memory, but stored within DATA statements. The DATA statements are part of a normal BASIC program, of course, and will SAVE and LOAD with no trouble at all.

We can now reconstruct our machine language program, placing it back into memory, with a simple BASIC POKE program:

```
80 FOR J=828 TO 847: READ X:POKE J,X:NEXT J
```

Now our program is safe and sound—it handles like BASIC, but it will do a machine language task for us as desired. Let's display the entire BASIC program

```
50 DATA    162,0,189,74,3,32,210,255,232,224,6
60 DATA    208,245,96,72,69,76,76,79,13
80 FOR    J =828 TO 847:READ X:POKE J,X:NEXT J
100 FOR J= 1 TO 3
110 SYS 828
120 NEXT J
```

This method of saving a machine language program is clean and trouble free, but it becomes awkward where long programs are involved. More advanced methods will be discussed in Chapter 6.

Things You Have Learned

—*Subroutines* can be called from machine language using the JSR command. There are several useful kernal subroutines permanently available.

—A BASIC program may call a machine language program as a subroutine:

the BASIC command is `SYS` The machine language subroutine returns to the calling point with an `RTS` (return from subroutine) instruction

—The `CHROUT` subroutine at address `$FFD2` allows output of a character, usually to the screen In addition to printable characters, special cursor- and color-control characters may be sent.

—Most machine language monitors have a small *assembler* to help program preparation, and a *disassembler* to assist in program checking.

—*Immediate mode* is signaled by use of the # symbol. The computer is asked to take the value given, instead of going to a specified address for its data.

—X and Y are called *index* registers. We may add the contents of X or Y to a specified address, to create an *effective address* that changes as the program runs. This addition is called *indexing*

—X and Y also have special instructions that increase or decrease the selected register by one These are called *increment* and *decrement* instructions, and are coded `INX`, `INY`, `DEX`, and `DEY`

Questions and Projects

Look through the table of ASCII characters in Appendix D. Note that hex `93` is "clear screen." Write a program to clear the screen and print `"HO HO!"`.

You may have noticed that in our example, we had register X counting up from zero to the desired value. What would happen if you started X at `5` and counted down? Try it if you like.

Remember that you can also include cursor movements, color codes (if your machine has color), and other special ASCII characters. Could you lay out the coding to draw a box? (Try it in BASIC first). Draw a box with the word `HELLO` inside it.
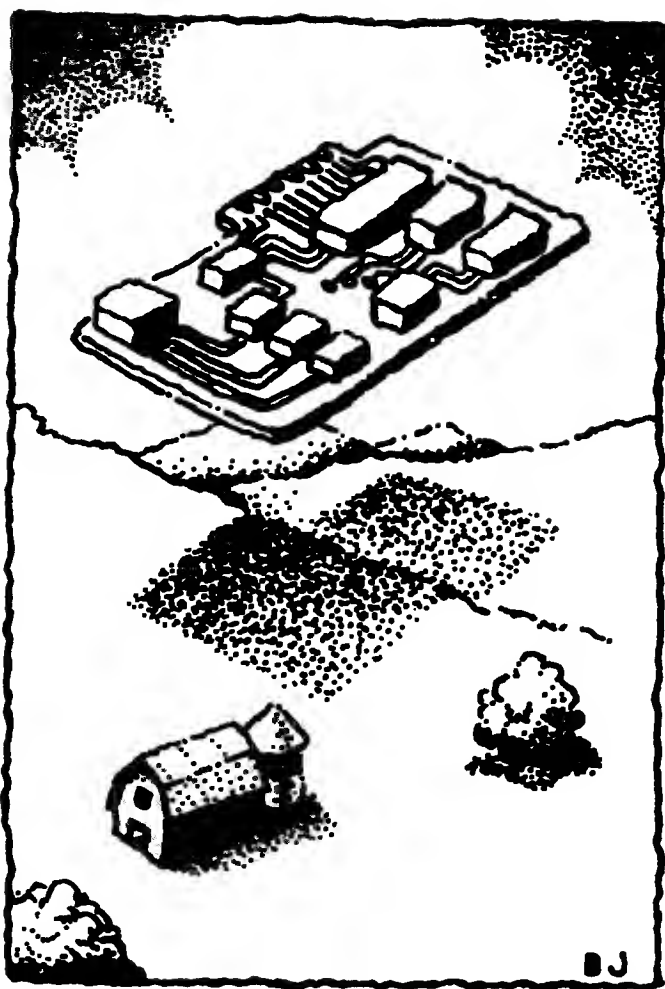
3

Flags, Logic, and Input

This chapter discusses:

- Flags that hold status information
- Testable flags Z, C, N, and V
- Signed numbers
- The status register
- First concepts of interrupt
- Logical operators OR, AND, EOR
- The GETIN subroutine for input
- The STOP subroutine

Flags

Near the end of Chapter 2, we coded a program that had the seemingly natural sequence

```
CPX #$06
BNE $....
```

It made sense: compare X for a value of 6, and if not equal, branch back. Yet it implies something extraordinary; the two instructions are somehow linked.

Let's flash forward for a moment. Even when you have a machine language program running, the computer "freezes" sixty times a second. The computer undertakes a special activity, called *interrupt processing*. It stops whatever it was doing, and switches to a new set of programs that do several tasks: flashing the cursor, checking the keyboard, keeping the clock up to date, and checking to see whether the cassette motor needs power. When it's finished, it "unfreezes" the main program and lets it continue where it left off.

This *interrupt* might take place between the two instructions shown above, that is, after the CPX and before the BNE. Hundreds of interrupt instructions might be executed between the two, yet nothing is harmed. The two instructions work together perfectly to achieve the desired effect. How can the computer do this?

The two instructions are linked by means of a *flag*—a part of the 650x that records that something has happened. The CPX instruction tests X and turns a special flag on or off to signal how the comparison turned out: equal or unequal. The BNE instruction tests that flag. If it's on (meaning equal), no branch will take place and the program will continue with the next instruction; if it's off (meaning not equal), a branch will take place.

In other words, some instructions leave a "trail" of status information; other instructions can check this information. The status information is called "flags." There are four flags that may be tested: Z, C, N, and V. They are discussed below.

Z *Flag*

The Z (zero) flag is probably misnamed, and should have been called the E flag (for "equals"). After any comparison (CPX to compare X, CPY to compare Y, or CMP to compare A), the Z flag will be set to "on" if the compared values are equal; otherwise it will be reset to "off."

Sometimes the Z flag checks for equal to zero, hence its name, Z for zero. This happens for every activity that may change one of the three data registers. Thus, any load command will affect the Z flag status. The same is true of increment and decrement instructions, which obviously change registers. And later, when we meet other operations such as addition and subtraction, they too will affect the Z flag.

There are many instructions that don't affect the Z flag (or any flag, for that matter). Store instructions (STA, STX, STY), never change a flag. Branch instructions test flags but don't change them.

An example will help illustrate the way that some instructions change flags and others do not. Examine the following coding:

```
LDA #$23    (Load 23 to A)
LDX #$00    (Load zero to X)
STA $1234   (store 23 to address $1234)
BEQ $....
```

Will the branch (BEQ) be taken, or will the 650x continue with the next instruction? Let's analyze the Z flag's activity step by step. The first instruction (LDA #$23) resets the Z flag, since 23 is not equal to zero. The second instruction (LDX #$00) sets the Z flag because of the zero value. The third instruction (STA $1234) does not affect the Z flag; in fact, store instructions do not affect any flags Thus, by the time we reach the BEQ instruction, the Z flag is set "on" and the branch will be taken.

650x reference manuals show the specific flags that are affected by each instruction. In case of doubt, they are easy to check.

The Z flag is quite busy—it clicks on and off very often since many instructions affect it. It's an important flag.

If the Z flag is set "on," the BEQ (branch equals) instruction will branch to the specified address; otherwise it will be ignored and the next instruction in sequence will be executed. If the Z flag is reset "off," the BNE (branch not equals) instruction will branch.

We can see in more detail how our program from Chapter 2 worked. CPX #$06 causes the Z flag to be set "on" if X contains the value 6; otherwise it causes the Z flag to be reset "off." BNE tests this flag, and branches back to the loop only if the Z flag is off—in other words, only if the contents of X is not equal to six.

C *Flag*

The C (carry) flag is probably misnamed, too. It should have been called the G E (greater/equal) flag, since after a comparison (C P X, C P Y, or C M P), the C flag is set "on" if the register (X, Y, or A) is greater than or equal to the value compared. If the register concerned is smaller, the C flag will be reset "off."

The C flag is not as busy as the Z flag. The C flag is affected only by comparison instructions and by arithmetic activities (add, subtract, and a type of multiplication and division called rotate or shift). When used in arithmetic, the C flag is properly named, since it acts as a "carry" bit between various columns as they are calculated. For example, an L D A instruction always affects the Z flag since a register is being changed, but never affects the C flag since no arithmetic or comparison is being performed.

If the C flag is set "on," the B C S (branch carry set) instruction will branch to the specified address; otherwise it will be ignored and the next instruction in sequence will be executed. If the C flag is reset "off," the B C C (branch carry clear) instruction will branch.

The C flag may be directly set or reset by means of the instructions S E C (set carry) and C L C (clear carry). We will use these instructions when we begin to deal with addition and subtraction.

If you examine the last program of Chapter 2, you will see that the B N E instruction could be replaced by B C C. Instead of "branch back if not equal to 6," we could code "branch back if less than 6." The operation would be the same in either case.

N *Flag*

The N (negative) flag is also probably misnamed. It should have been called the H B (high bit) flag, since numbers are positive or negative only if they are used in a certain way. The N flag is set to indicate that a register has been given a value whose high bit is set.

The N flag is as busy as the Z flag; it changes with every instruction that affects a register. The N flag is affected by comparisons, but in this case its condition is not usually meaningful to the programmer.

To sort out the operation of the N flag, it's important to become familiar with hexadecimal-to-binary conversion. For example, will L D A #$b5 set the N flag? Rewrite it into binary: $b5 equals %01100101. We can see

that the high bit is not set, meaning that the N flag will be off after loading this value. As another example, suppose we LDX #$DA. Hex DA is 11011010 binary. We see that the high bit is on and thus the N flag is set.

If the N flag is set "on," the BMI (branch minus) instruction will branch to the specified address; otherwise it will be ignored and the next instruction in sequence will be executed. If the N flag is reset "off," the BPL (branch plus) instruction will branch

A Brief Diversion: Signed Numbers

How can a location—which is usually thought to contain a decimal value from 0 to 255—contain a negative number? It's up to the programmer to decide whether a memory value is *unsigned*, having a value range from 0 to 255, or *signed*, having a value range from −128 to +127. There are still a total of 256 possibilities. The computer's memory simply holds *bits*, while the programmer decides how the bits are to be used in a specific case

Mathematically, it's described this way: signed numbers, if desired, are held in *two's-complement* form. We can hold −1 as hex FF, and −2 as hex FE, all the way down to −128 as hex 80. You may have noticed that in all the examples, the high bit is set for these negative numbers.

We may need more intuitive help, however. If the computer loads the decimal value 200 into the A register with LDA #$C8, the N flag will be set and will seemingly indicate that 200 is a negative number. It may be more comfortable to simply think of 200 as a number with the high bit set. But in a sense, 200 could be a negative number if we wanted it to be. Let's examine the situation by means of examples.

If I were asked to count down in hexadecimal from 10, I'd start out $10, $0F, $0E, and $0D, continuing down to $02, $01, and $00. If I needed to keep going, I'd continue past $00 with $FF; in this case, hex FF would clearly represent negative one. Continuing, FE, FD, and FC would represent −2, −3, and −4. And the high bit is set on all these "negative" numbers.

Let's discuss a decimal analogy. Suppose you have a cassette recorder with a counter device attached, and the counter reads 0025. If you rewind the unit a distance of 30 units, you would not be surprised to see a value of 9995 on the counter and would understand that it meant a position of −5. If you had a car with 1,500 miles on the odometer, and "rolled back"

the mileage by 1,501 miles, you'd see a reading of 99999, which would mean − 1. (The author does not know this from personal experience, but is assured by many machine language students that it is so.) In these cases. based on the decimal system, the negative numbers are called "ten's complement."

V *Flag*

As with the other flags, the V (overflow) flag is probably misnamed. It should have been called the S A O (signed arithmetic overflow) flag, since it is affected only by addition and subtraction commands, and is meaningful only if the numbers concerned are considered to be signed.

The V flag is used only occasionally in typical 650x coding. Many machine language programs don't use signed numbers at all. The most typical use of the V flag is in conjunction with a rather specialized command, B I T (bit test). For this instruction, the V flag signals the condition of bit 6 of the memory location being tested. In this case, V and N work in a similar way: N reflects the high bit, bit 7, and V represents the "next bit down," bit 6. The B I T command is used primarily for testing input/output ports on I A (interface adaptor) chips.

If the V flag is set "on," the B V S (branch overflow set) instruction will branch to the specified address; otherwise it will be ignored and the next instruction in sequence will be executed. If the V flag is reset "off," the B V C (branch overflow clear) instruction will branch.

The V flag may be directly reset by means of the C L V (clear overflow) instruction. Oddly, there is no equivalent instruction to set the flag.

One special feature of the V flag: on some 650x chips, the V flag can be set by hardware. There is a pin on the chip that can be used so that an external logic signal will trigger the V flag.

A Brief Diversion: Overflow

The term *overflow* means "the result is too big to fit." For example, if I add 200 to 200, the total is 400 . . . but this won't fit in a single byte. If we have only a single byte to store the result, we say that the addition has encountered overflow, and we can't produce a meaningful answer.

If we are using unsigned numbers, the C flag tells us about overflow. If we are using signed numbers, V tells the story. We'll take this up again in the next chapter.

Flag Summary

A brief table may help review the four testable flags.

Flag Name	Brief Meaning	Activity Level	Branch Taken If: Set	Not-Set
Z	Zero, equal	Busy	BEQ	BNE
C	Carry, greater/equal	Quiet	BCS	BCC
N	Negative, high-bit	Busy	BMI	BPL
V	Signed arithmetic overflow	Quiet	BVS	BVC

The Status Register

The preceding flags—and three others—may be viewed within the status register (SR). You may recall that the machine language monitor gives an SR display. If you know how to read it, you can see the condition of all flags.

Each flag is a bit within the status register. Again, it's useful to be able to easily translate the hexadecimal display, so as to view the individual flags. Here's a chart of the flags within the status register:

```
7 6 5 4 3 2 1 0
N V - B D I Z C
```

Taking the bits one at a time, starting at the high bit:

N—the N flag, as above

V—the V flag, as above.

Bit 5—unused You'll often find that this bit is "on."

B—"Break" indicator. When an interrupt occurs, this signals whether or not the interrupt was caused by a BRK instruction.

D—Decimal mode indicator. This changes the manner in which the add and subtract instructions operate. In Commodore machines, this flag will always be off. Don't turn it on unless you know exactly what you're doing. This flag may be turned on with the SED (set decimal) instruction, and turned off with the CLD (clear decimal) instruction.

I—Interrupt disable. More exactly, this bit disables the IRQ (interrupt request) pin activity. More on this control bit much later. This flag may be turned on with the SEI (set interrupt disable) instruction, and turned off with the CLI (clear interrupt disable) instruction

Z—the Z flag, as above.

C—the C flag, as above.

Flags B, D, and I are not testable flags in that there are no branch instructions that test them directly D, the decimal mode flag, and I, the interrupt lockout flag, may be considered "control" flags. Instead of reporting conditions found as the program runs, they control how the program operates.

When we see a value displayed in the SR, or status register, we may examine it to determine the condition of the flags, especially the testable flags Z, C, N, and V. For example, if we see an SR value of $B1, we translate to binary %10110001 and know that the N flag is on, the V flag is off, the Z flag is off, and the C flag is on.

You may change these flags by typing over the displayed value in the machine language monitor Be careful you don't accidentally set the D or I flags

A Note on Comparison

If we wish to compare two bytes with each other, we must perform a comparison. One value must be in a register (A, X, or Y); the other must either be stored in memory, or must be an immediate value we use in the instruction.

We will use the appropriate compare instruction depending on the register involved; CMP for the A register, CPX for the X register, and CPY for the Y register Following the comparison, we may use any of the following branch tests:

BEQ—branches if the two bytes are equal

BNE—branches if the two bytes are not equal

BCS—branches if the value in the register is greater than or equal to the other value

BCC—branches if the value in the register is less than the other value.

We can use more than one branch instruction after a comparison. Suppose our program wanted to test the Y register for a value equal to or less than 5. We might code

```
CPY #$05
BEQ ..somewhere
BCC ..somewhere
```

We can see that our code will branch if the value is equal to 5 (using the BEQ) or less than 5 (using the BCC); otherwise it will continue without branching. In this case, we could make the coding more efficient by changing it to read

```
CPY #$06
BCC ..somewhere
```

A little common sense will tell us that testing a number to see if it is less than 6 is the same as testing it to see if it is less than or equal to 5. Common sense is a valuable programming tool.

Instructions: A Review

We have looked at the three data registers—A, X, and Y—and have seen three types of operation we can perform with them:

Load:	LDA, LDX, LDY
Store:	STA, STX, STY
Compare:	CMP, CPX, CPY

Up to this point, the registers have identical functions, and we can use any of them for any of these functions But new instructions are creeping in that give a different personality to each of the three.

We have noted that INX, INY. DEX. and DEY for increment and decrement are restricted to X and Y only; and we've also mentioned that X and Y can be used for indexing. Soon, we'll start to examine some of the functions of the A register, which is often called the *accumulator* because of its ability to do arithmetic.

We have seen JSR, which allows us to call a subroutine of prewritten instructions. We've used RTS, which says, "Go back to the calling point," even if the calling point is a BASIC program. And we've almost abandoned the BRK instruction, which stops the program and goes to the machine language monitor BRK will be useful in checking out programs. Specifically, we can stop a program at any time by inserting a BRK instruction, allowing us to see whether the program is behaving correctly and whether it has done the things we planned.

There are eight branch instructions. They have already been discussed, but there is one additional piece of information that is important to keep in mind. All branches are good only for short hops of up to a hundred memory locations or so. So long as we write short programs, that won't be a limitation; but we'll look at this more closely in Chapter 5.

Logical Operators

Three instructions perform what are called *logical* operations. They are: AND (Logical AND); ORA (Logical OR); and EOR (Exclusive OR). These instructions work on the A register only.

Mathematicians describe these operations as *commutative*. For example, a value of $3A "AND" $57 gives exactly the same result as $57 "AND" $3A. The order doesn't matter. But we often use these functions—and think of them—in a particular order. It's the same as with addition, where we think of a "total" to which is added an "amount" to make a "new total." With the logical operators we often think of a "value," which we manipulate with a "mask" to make a "modified value."

Logical operators work in such a way that each bit within a byte is treated independently of all the other bits. This makes these instructions ideal for extracting bits, or manipulating certain bits while leaving others alone.

We'll look at formal definitions, but the following intuitive concepts are useful to programmers:

AND—turns bits off.

ORA—turns bits on

EOR—flips bits over.

AND—*Logical* AND *to* A

For each bit in the A register, AND performs the following action:

Original A Bit	Mask	Resulting A Bit
0	0	0
1	0	0
0	1	0
1	1	1

Examine the upper half of this table. When the mask is zero, the original bit in A is changed to zero. Examine the lower half. When the mask is one, the original bit is left unchanged. Hence, AND can selectively turn bits off.

Example: Turn off bits 4, 5, and 6 in the following value: $C7

Original value:		11000111
Mask.	AND	10001111 (hex 8F)
		————————
Result		10000111
		xxx

Note that the bits marked have been forced to "off," while all other bits remain unchanged.

ORA—*Logical* OR *to* A

For each bit in the A register, ORA performs the following action:

Original A Bit	Mask	Resulting A Bit
0	0	0
1	0	1
0	1	1
1	1	1

Examine the upper half of this table. When the mask is zero, the original bit in A is left unchanged. Examine the lower half. When the mask is one, the original bit is forced to "on." Hence, ORA can selectively turn bits on.

Example: Turn on bits 4, 5, and 6 in the following value: $C7

```
Original value:      11000111
Mask.          EOR   01110000 (hex 70)
                     --------
Result               11110111
                      x x x
```

Note that the bits marked have been forced to "on," while all other bits remain unchanged.

EOR—*Exclusive* OR *to* A

For each bit in the A register, EOR performs the following action:

Original A Bit	Mask	Resulting A Bit
0	0	0
1	0	1
0	1	1
1	1	0

Examine the upper half of this table. When the mask is zero, the original bit in A is left unchanged. Examine the lower half. When the mask is one, the original bit is inverted; zero becomes one and one becomes zero. Hence, EOR can selectively flip bits over.

Example: Invert bits 4, 5, and 6 in the following value: $C7

```
Original value:      11000111
Mask:          EOR   01110000 (hex 70)
                     --------
Result               10110111
                      x x x
```

Note that the bits marked have been flipped to the opposite value, while all other bits remain unchanged.

Why Logical Operations?

We use these three commands—AND, ORA, and EOR—to change or control individual bits within a byte of information. The commands are unusual in that each bit may be manipulated independently of the others.

We don't seem to be working with numbers when we use these commands. Rather, we're working with each individual bit, turning it on or off as we wish.

Why would we turn individual bits on or off? There are several possible reasons. For example, we might wish to control external devices through the IA's (interface adaptors). Within the IA's input and output ports each of the eight bits might control a different signal; we might want to switch one control line on or off without affecting other lines.

When we're looking at input from an IA port, we often read several input lines mixed together within a byte. If we want to test a specific bit to see if it is on or off, we might mask out all other bits with the AND instruction (changing unwanted bits to zero); if the remaining bit is zero, the whole byte will now be zero and the Z flag will be set.

Why would we want to flip bits over? Many "oscillating" effects—screen flashing or musical notes—can be accomplished this way.

Finally, the logical operators can be useful in code translation. For example, here are the values for ASCII 5 and binary 5.

        ASCII   %00110101
        Binary   %00000101

We must use the ASCII value for input or output. We must use the binary value for arithmetic, particularly addition and subtraction. How could we get from one to the other? By taking bits out (AND) or putting bits in (ORA). Alternatively, we could use addition or subtraction; the logical operators, however, are simpler.

Input: The GETIN Subroutine

We have seen how we can use CHROUT at $FFD2 to produce output to the screen. Now we'll look at the input side—how to use the GETIN subroutine at $FFE4 to get characters from the keyboard buffer.

You may be familiar with the GET statement in BASIC. If so, you'll find the same characteristics in GETIN:

- Input is taken from the keyboard buffer, not the screen.
- If a key is held down, it will still be detected once only.
- The subroutine returns immediately
- If no key is found, a binary zero is returned in A
- If a key is found. its ASCII value will be in A.
- Special keys, such as RETURN, RVS, or color codes, will be detected

To call for a key from the keyboard, code JSR $FFE4. Values in X and Y are not guaranteed to be preserved, so if you have important information in either register, put it away into memory.

Subroutine·   GETIN
Address. $FFE4
Action: Takes a character from the input channel and places it into the A register The input channel is the keyboard input buffer unless arrangements have been made to switch it

The character received is usually ASCII (or PET ASCII). When read from the keyboard, the action is similar to a BASIC GET statement: one character will be taken from the buffer; it will not be shown on the screen. If no character is available from the keyboard input buffer, a value or binary zero will be put into the A register. The subroutine will not wait for a key to be pressed but will always return immediately.

Registers: The A register will of course always be affected. X and Y are likely to be changed; do not have data in these when calling GE-TIN.

Status: Status flags may be changed. In the VIC and Commodore 64,

---

If we want keyboard input to appear on the screen, we should follow a call to GETIN, $FFE4, with a call to CHROUT, $FFD2, so that the received character is printed.

STOP

Machine language programs will ignore the RUN/STOP key   . unless the program checks this key itself. It may do so with a call to STOP, address $FFE1. This checks the RUN/STOP key at that moment. To make the key operational, $FFE1 must be called frequently.

A call to FFE1 should be followed by a BEQ to a program exit so that the program will terminate when RUN/STOP is pressed.

The RUN/STOP key is often brought into play while programs are being tested, so that unexpected "hangups" can still allow the program to be terminated. Coding to test the RUN/STOP key is often removed once testing is complete, on the assumption that no one will want to stop a perfect program. Incidentally, if you plan to write nothing but 100 percent perfect programs, you will not need to use this subroutine.

Subroutine	STOP
Address	$FFE1
Action.	Check the RUN/STOP key. If RUN/STOP is being pressed at that instant, the Z flag will be set when the subroutine returns.

In PET/CBM, the system will exit to BASIC and say READY if the RUN/STOP key is being pressed. In this case, it will not return to the calling machine language program.

Registers: A will be affected. X will be affected only if the RUN/STOP key is being pressed.

Status: Z signals whether RUN/STOP is being pressed.

Programming Project

Here's our task: we wish to write a subroutine that will wait for a numeric key to be pressed. All other keys (except RUN/STOP) will be ignored.

When a numeric key is pressed, it will be echoed to the screen, and then the subroutine will be finished. One more thing. The numeric character will arrive in ASCII from the keyboard: we wish to change it to a binary value before giving the final RTS statement. This last operation has no useful purpose yet, except as an exercise, but we'll connect it up in the next chapter.

Coding sheets ready? Here we go.

```
.A 033C   JSR   $FFE1
```

We will check the RUN/STOP key first. But wait. Where will we go if we find that the key is pressed? To the RTS, of course; but we don't know where that is, yet. In these circumstances, we usually make a rough guess and correct it later. Make a note to check this one . . .

```
.A 033F   BEQ   $0351
.A 0341   JSR   $FFE4
```

Now we've gotten a character; we must check that it's a legitimate numeric. The ASCII number set 0 to 9 has hexadecimal values $30 to $39. So if the value is less than $30, it's not a number. How do we say "less than?" After a compare, it's BCC (branch carry clear). So we code

```
.A 0344   CMP   #$30
.A 0346   BCC   $033C
```

Did you spot the use of immediate mode at address $0344? Make sure you follow the logic on this. Another point: what if no key has been pressed? We're safe. There will be a zero in the A register, which is less than hex 30; this will cause us to go back and try again.

Now for the high side. If the number is greater than hex 39, we must reject it since it cannot be an ASCII numeric. Our first instinct is to code CMP #$39 and BCS. But wait! BCS (branch carry set) means "branch if greater than or equal to." Our proposed coding would reject the digit 9, since the carry flag would be set when we compared to a value of hex 39.

We must check against a value that is one higher that $39. Be careful, though, for we're in hexadecimal. The next value is $3A. Code it:

```
.A 0348   CMP   #$3A
.A 034A   BCS   $033C
```

If we get this far, we must have an ASCII character from 0 to 9; let's print it to the screen so that the user gets visual feedback that the right key has been pressed:

```
.A 034C   JSR   $FFD2
```

Now for our final task. We are asked to change the ASCII character into true binary. We may do this by knocking off the high bits. We remember, of course, that to turn bits off we must use AND:

```
.A 034F   AND   #$0F
.A 0351   RTS
```

It's a good thing that we printed the character first, and then converted to binary; the character must be ASCII to print correctly.

One last thing. We had a branch (on the RUN/STOP key) that needed to connect up with the RTS. Did you make that note about going back and fixing up the branch? Now is the time to do it, but before you go back,

terminate the assembly with an extra RETURN on the keyboard (the assembler gets confused if it prompts you for one address and you give another; get out before you go back).

By a fortunate stroke of luck, we happen to have guessed the right address for the BEQ at address $033F. But if we hadn't, you know how to change it, don't you?

Check your coding, disassemble, go back to BASIC and run with a SYS 828. Tap a few letter keys and note that nothing happens. Press a number, and see it appear on the screen. The program will terminate. SYS it again and see if the RUN/STOP works. Try a BASIC loop to confirm that BASIC and machine language work together.

Project for enthusiasts Try modifying the program so that it checks for alphabetic characters only. Alphabetic characters run from $41 to $5A, inclusive.

Things You Have Learned

—*Flags* are used to link instructions together. This might be an activity such as load or compare, followed by a test such as branch on a given condition

—Some instructions affect one or more flags, and some do not affect flags. Thus, an instruction that sets a flag might not be followed immediately with the instruction that tests or uses that flag

—There are four *testable* flags· Z (zero, or equals); C (carry, or greater/equal), N (negative, or high bit), and V (signed arithmetic overflow). The flags are checked by means of "branch" instructions such as BEQ (branch equal) or BNE (branch not equal)

—Flags are stored in the *status register*, sometimes called the *processor status word* The SR contains the four testable flags, plus three other flags: B (break indicator), D (decimal mode for add/subtract); and I (interrupt lockout). The hexadecimal value in SR can be changed to binary and used to determine the exact condition of all flags.

—Usually, the processor is *interrupted* sixty times a second to do special high-priority jobs. Everything, including the status register flags, is carefully preserved so that the main program can continue as though nothing had happened.

—A number stored in memory can be considered as *signed* if we decide to handle it that way. The value of a signed number is held in *two's-complement* form The high bit of the number is zero if the number is positive, one if the number is negative The computer doesn't care It handles the bits whether the number is considered signed or not, but we must write our program keeping in mind the type of number being used.

—There are three *logical operator* instructions· AND, ORA, and EOR These allow us to modify bits selectively within the A register AND turns bits off, ORA turns bits on; and EOR inverts bits, or flips them over.

Questions and Projects

Write extra coding to allow both numeric and alphabetic characters, but nothing else.

Write a program to accept only alphabetic characters. As each ASCII character is received, turn on its high bit with ORA #$80 and then print it. How has the character been changed?

Write a program to accept only numeric digits. As each ASCII character is received, turn off its lowest bit with AND #$FE and then print it What happens to the numbers? Can you see why?
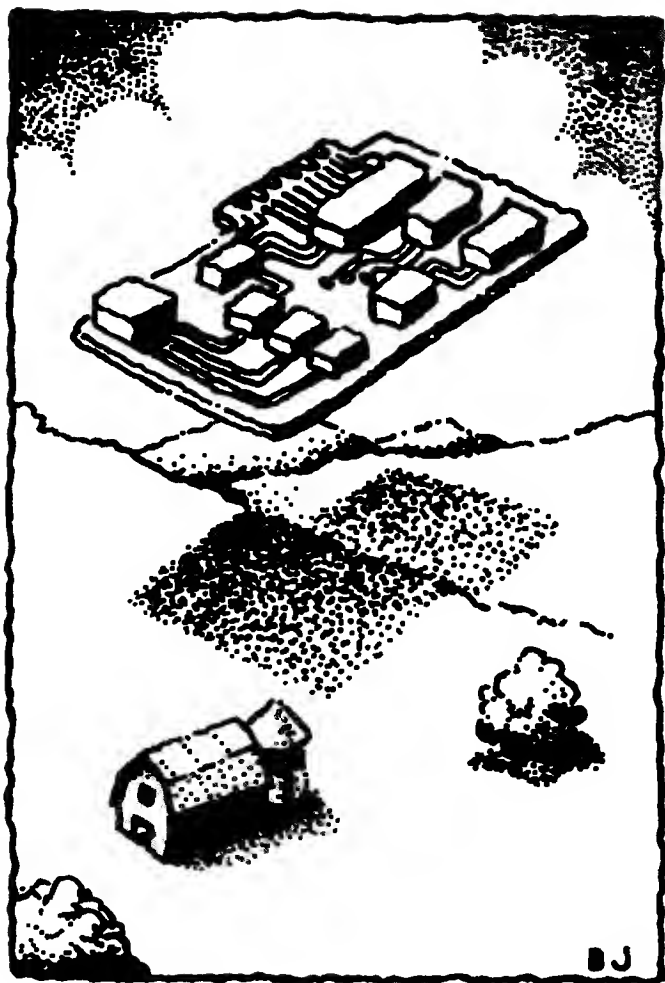
4

Numbers,
Arithmetic,
and
Subroutines

This chapter discusses:

- Numbers: signed and unsigned
- Big numbers: multiple bytes
- Arithmetic: add and subtract
- Rotate and shift instructions
- Multiplication
- Home grown subroutines

Numbers: Signed and Unsigned

We have looked briefly at the question of signed versus unsigned numbers. The most important concept is that you, the programmer, choose whether or not a number is to be considered a signed number (for a single byte, in the decimal range -128 to $+127$) or an unsigned integer (single-byte range 0 to 255).

It makes no difference to the computer. If you consider a number signed, you may wish to test the sign using the N flag. If not, you won't do such a test.

Big Numbers: Multiple Bytes

You may use more than one byte to hold a number. Again, it's your decision. If you think the numbers may go up to a million, you might allocate three bytes (or more or fewer). If you are doing arithmetic on multi-byte numbers, the computer will help you by signaling in the carry flag that there's something to be carried across from a lower byte to a higher one. But it's up to you to write the code to handle the extra bytes.

You may size numbers by using the following table:

	Unsigned.	Signed.
1 byte	0 to 255	-128 to $+127$
2 bytes	0 to 65,535	-32768 to $+32767$
3 bytes	0 to 16,777,215	$-8,388,608$ to $+8,388,607$
4 bytes	to over 4 billion	-2 billion to $+2$ billion

It's possible to work with binary fractions, but that is beyond the scope of this book. Many applications "scale" numbers, so that dollar-and-cents amounts are held as integer quantities of pennies. Thus, two bytes unsigned would hold values up to $655.35, and three bytes up to $167,772.15.

When signed numbers are held in multiple bytes, the sign is the highest bit of the highest byte only.

We will concentrate on single-byte arithmetic principles here, touching on multiple-byte numbers as a generalization of the same ideas.

Addition

Principles of addition are similar to those we use in decimal arithmetic; for decimal "columns," you may substitute "bytes." Let's look at a simple decimal addition:

```
  1Ⴣ28ႽᏮ
+ Ꝫ89217
  ------
```

Rule 1. We start at the right-hand column (the low-order byte).

Rule 2· We add the two values, plus any carry from the previous column. A new carry may be generated, it can never be greater than one (ADC includes any carry from a previous activity, and may generate a new carry bit, which is either 0 or 1)

Rule 3. When we start at the right-hand column, there is no carry for the first addition (We must clear the carry with CLC before starting a new addition.)

Rule 4 When we have finished the whole addition, if we have a carry and no column to put it in, we say the answer "won't fit." (If an addition sequence of unsigned numbers ends up with the carry flag set, it's an overflow condition.)

```
     HIGH BYTE          LOW BYTE

                                          START:
                                         NO CARRY
     00101011          10111001          /
+    00001010          11100101         /
     --------          --------
                       10011110
                        /
                       /
           CARRY     /
                   /
     00110110
```
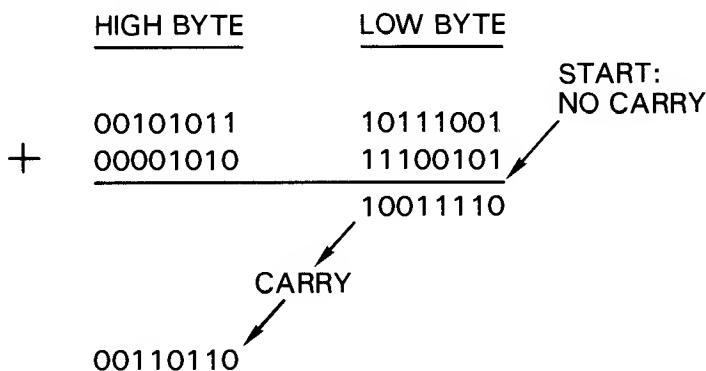
Figure 4.1

How do we translate these rules into machine language addition?

1 Before we start an addition sequence, clear the carry with CLC.

2. If the numbers are more than one byte in size, start at the low byte and work up to the high ones Addition will take place in the A register only; you may add the contents of an address or an immediate value. The carry flag will take care of any carries.

3. When the addition sequence is complete, check for overflow:
   a) if the numbers are unsigned, a set C flag indicates overflow;
   b) if the numbers are signed, a set V flag indicates overflow.

Thus, to add two unsigned numbers located at addresses $0Ꝫ80 and $0Ꝫ81 and to place the result at $0Ꝫ82, we might code

```
CLC
LDA $0380
ADC $0381
STA $0382
```

We might also BCS to an error routine, if desired.

To add a two-byte number located at $03A0 (low) and $03A1 (high) to another two-byte number located at $03B0 (low) and $03B1 (high), placing the result at $03C0/1, we might code

```
CLC
LDA $03A0
ADC $03B0
STA $03C0
LDA $03A1
ADC $03B1
STA $03C1
```

Again, we might BCS to an overflow error routine

If we had two-byte *signed* numbers in the same locations, we'd add them exactly the same way, using the same code as above. In this case, however, we'd check for overflow by adding the instruction BVS, which would branch to an error routine The carry flag would have no meaning at the end of the addition sequence.

Subtraction

Subtraction might be defined as "upside down" addition. The carry flag again serves to link the parts of a multibyte subtraction, but its role is reversed. The carry flag is sometimes called an "inverted borrow" when used in subtraction. Before performing a subtraction, we must set the C flag with SEC. If we are worried about unsigned overflow, we look to confirm that the carry is set at the completion of the subtraction operation. If the carry is clear, there's a problem.

Thus, to perform a subtraction, we follow these rules·

1 Before we start a subtraction sequence, set the carry with SEC

2. If the numbers are more than one byte in size, start at the low byte and work up to the high ones Subtraction will take place in the A register only; you may subtract the contents of an address or an immediate value The C flag will take care of any "borrows."

3 When the subtraction sequence is complete, check for overflow
   a) if the numbers are unsigned, a clear C flag indicates overflow;

b) if the numbers are signed, a set V flag indicates overflow.

Thus, to subtract two unsigned numbers located at addresses $0380 and $0381 and to place the result at $0382, we might code

```
SEC
LDA $0380
SBC $0381
STA $0382
```

A BCC could go to an error routine

Comparing Numbers

If we have two unsigned numbers and wish to know which one is larger, we can use the appropriate compare instruction—CMP, CPX, or CPY— and then check the carry flag. We've done this before. If the numbers are more than one byte long, however, it's not quite so easy. We must then use a new technique.

The easiest way to go about such a comparison is to subtract one number from the other. You need not keep the result, all you care about is the carry flag when the subtraction is complete. If the C flag is set, the first number (the one you are subtracting *from*) is greater than or equal to the second number. Why? Because carry set indicated that the unsigned subtraction was legal; we have subtracted the two numbers and have obtained a positive (unsigned) result. On the other hand, if the C flag ends up clear, this would mean that the first number is less than the second. The subtraction couldn't take place correctly since the result—a negative number—can't be represented in unsigned arithmetic.

Left Shift: Multiplication by Two

If we write the decimal numbers 100 and 200 in binary, we see an interesting pattern:.

```
100:   %01100100
200:   %11001000
```

To double the number, each bit has moved one position to the left. This makes sense, since each bit has twice the numeric "weight" of the bit to its right.

The command to multiply a byte by two is ASL (arithmetic shift left). A zero bit is pushed into the low (or "right") side of the byte; all bits move

left one position; and the bit that "falls out" of the byte—in this case, a zero bit—moves into the carry. It can be diagrammed like this:

CARRY
(C FLAG)

ASL

IN AN ASL (ARITHMETIC SHIFT LEFT), EACH BIT
MOVES ONE POSITION LEFT. A ZERO MOVES INTO THE
LOW-ORDER BIT.

Figure 4.2

That's good for doubling the value of a single byte. If a "one" bit falls into the carry flag, we can treat that as an overflow. What about multiple bytes?

It would be ideal if we had another instruction that would work just like ASL. Instead of pushing a zero bit into the right hand side of the byte, however, it would push the carry bit, that is, the bit that "fell out" of the last operation. We have such an instruction: ROL

ROL (rotate left) works exactly like ASL except that the carry bit is pushed into the next byte. We can diagram it as follows:
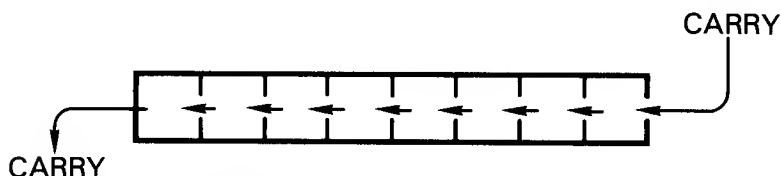
CARRY

CARRY

IN A ROL (ROTATE LEFT), THE CARRY MOVES INTO
THE LOW ORDER BIT; EACH BIT MOVES LEFT; AND THE
HIGH ORDER BIT BECOMES THE NEW CARRY.

Figure 4.3

Thus, we can hook two or more bytes together. If they hold a single multibyte number, we can double that number by starting at the low-order end. We ASL the first value and ROL the remainder. As the bits fall out of each byte, they will be picked up in the next.

Multiplication

Multiplying by two may not seem too powerful. We can build on this starting point, however, and arrange to multiply by any number we choose.
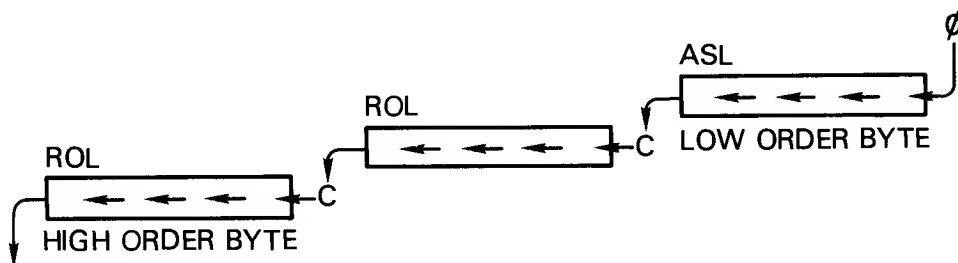
TO MULTIPLY A THREE-BYTE NUMBER BY TWO, WE
SHIFT THE LOW ORDER BYTE WITH ASL; THEN WE USE
ROL TO ALLOW THE C FLAG TO "LINK" FROM ONE
BYTE TO THE NEXT.

Figure 4.4

We won't deal with a generalized multiplication routine here, but a couple of specific examples can be shown.

How can we multiply by four? Multiply by two, twice. How can we multiply by eight? Multiply by two, three times.

Here's an important one We often want to multiply by ten. For example, if a decimal number is being typed in at the keyboard, the number will arrive one digit at a time. The user might type 217, for example The program must then input the two and put it away; when the one arrives, the two must be multiplied by ten, giving twenty, and the one added; when the seven is typed, the twenty-one must be multiplied by ten before the seven is added. Result 217 in binary. But we must first know how to multiply by ten.

To multiply by ten, you first multiply by two; then multiply by two again. At this point, we have the original number times four. Now, add the original number, giving the original number times five. Multiply by two one last time and you've got it. We'll see an example of this in Chapter 7

Right Shift and Rotate: Dividing by Two

If we can multiply by two by shifting (and rotating) left, we can divide by two by moving the bits the other way. If we have a multibyte number, we must start at the high end.

LSR (logical shift right) puts a zero into the left (high-order) bit, moves all the bits over to the right, and drops the leftover bit into the carry. ROR (rotate right) puts the carry bit into the left bit, moves everything right, and
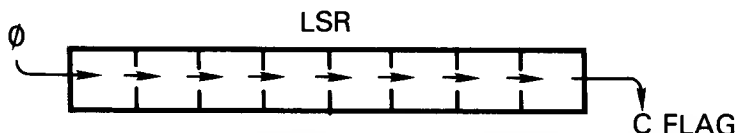
IN AN LSR, ZERO MOVES INTO THE HIGH BIT, AND ALL
BITS MOVE RIGHT ONE POSITION; THE LOWEST BITS
BECOME THE CARRY.
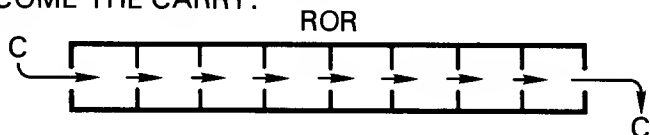
IN A ROR, THE CARRY MOVES INTO THE HIGH BIT AND
ALL BITS MOVE RIGHT ONE POSITION; THE LOWEST
BIT BECOMES THE NEW CARRY.
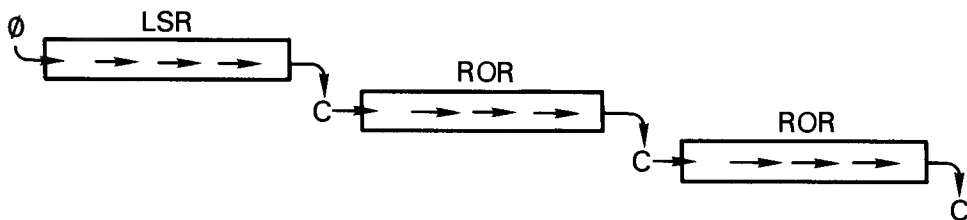
TO DIVIDE A THREE-BYTE NUMBER BY TWO, WE SHIFT
THE HIGH-ORDER BYTE WITH LSR; THEN WE USE ROR
TO ALLOW THE C FLAG TO "LINK" FROM BYTE TO
BYTE.

Figure 4.5

drops the leftover bit into the carry once again. At the end of a right-shifting
sequence, the final carry bit might be considered a *remainder* after dividing
by two.

Comments on Shift and Rotate

As you might expect of arithmetic instructions, the shift and rotate instruc-
tions normally operate in the A register. But there's an extra bonus: these
instructions also can operate directly on memory. In other words, the
computer can go to any address in memory and shift the bits at that address
directly, without loading the data into a register.

For this reason, you'll often see the instructions coded with the identity of
the A register coded in the address part of the instruction. We would code

LSR A so as to distinguish from LSR $1234, where the contents of memory is being shifted.

When a rotate or shift is performed directly on a memory location, the Z, N, and C flags are affected according to the contents of memory. Z will be set if the contents of the location ends up as zero; N if the high bit is set; and C performs its standard role of catching the leftover bit.

Some programmers wonder about the terms *logical* and *arithmetic*, used as part of the definition. The distinction is related to the way that signed numbers are treated. "Logical" means that the sign of a number will probably be lost if the number was intended to be signed "Arithmetic" means that the sign will probably be preserved. It's purely a terminology question: the bits themselves move exactly as you would expect them to do.

Subroutines

We have written programs that are subroutines called by BASIC We have written subroutine calls to built-in operations such as $FFD2 or $FFE4. Can we also write our own subroutine and arrange to call it?

Of course we can. RTS (return from subroutine) does not mean "return to BASIC." It means "return to whoever called this routine." If BASIC called up the machine language routine, RTS takes you back to BASIC. If another machine language program called up the subroutine, RTS will return to the calling point.

We wrote a useful subroutine in the last chapter. Its purpose was to accept only numeric keys, echo them to the screen, and convert the ASCII value to binary Now we'll use this subroutine to build a more powerful program. Here it is. Be sure it's entered in your computer.

```
.A 033C  JSR  $FFE1
.A 033F  BEQ  $0351
.A 0341  JSR  $FFE4
.A 0344  CMP  #$30
.A 0346  BCC  $033C
.A 0348  CMP  #$3A
.A 034A  BCS  $033C
.A 034C  JSR  $FFD2
.A 034F  AND  #$0F
.A 0351  RTS
```

The Project

Here is our mission: using the above subroutine, we wish to build a simple addition program. Here's how we want it to work. The user will touch a numeric key, say "3". Immediately, "3 + " will appear on the screen. Now the user will touch another key, say "4", and the program will complete the addition so that the screen shows "3 + 4 = 7". We will assume that the total is in the range 0 to 9 so that we don't have to worry about printing a two-digit answer—don't try 5 + 5 or you'll get a wrong answer

Here we go. We must start our coding at address $0352 so as not to disturb our subroutine. We'll need to give SYS 850 to make this one go.

```
. A 0352 JSR $033C
```

We call our prewritten subroutine, which waits for a numeric key, echos it to the screen, and converts the value to binary in the A register.

Our next action is to print the plus sign We know how to do this, once we look up the ASCII code for this character. Appendix D tells us that it's $2B, so we'll need to LDA #$2B and JSR $FFD2. But wait a minute! Our binary value is in the A register, and we don't want to lose it. Let's store the value somewhere:

```
. A 0355   STA   $03C0
. A 0358   LDA   #$2B
. A 035A   JSR   $FFD2
. A 035D   JSR   $033C
```

We picked $03C0, since nobody seems to be using it, and put the binary number safely away there. Now we print the plus sign, and go back to ask for another digit.

When the subroutine returns, it has a new binary value in the A register; the digit has been neatly printed on the screen behind the plus sign. Now we need to print the equal sign. But again, wait! We must put our binary value away first.

We could place the value into memory—perhaps $03C1 would do—but there's another way. We don't seem to be using X or Y for anything at the moment, so let's slip the value across into one or the other. We have four "transfer" commands that will move information between A and either index register:

TAX—Transfer A to X   TAY—Transfer A to Y
TXA—Transfer X to A   TYA—Transfer Y to A

Like the load series of commands, these instructions make a copy of the information. Thus, after TAX, whatever information was in A is now also in X. Again like the load commands, the Z and N status flags are affected by the information transferred. It doesn't matter whether we use X or Y. Let's pick X.

```
.A 0360   TAX
.A 0361   LDA #$3D
.A 0363   JSR $FFD2
```

We have put our second value into X and printed the equal sign ($3D). Now we can bring the value back and do our addition. The next two instructions can come in any order:

```
.A 0366   TXA
.A 0367   CLC
.A 0368   ADC $03C0
```

We have our total in the A register. It's almost ready to print, except for one thing: it's in binary. We want it in ASCII.

Assuming the total is in the range 0 to 9, we can convert it directly to a single ASCII digit with an ORA operation. (If it's greater than nine, you're cheating and the answer won't make sense.)

```
.A 036B   ORA   #$30
.A 036D   JSR   $FFD2
```

Are you basically a neat person? Then you'll want to print a RETURN to start a new line:

```
.A 0370   LDA   #$0D
.A 0372   JSR   $FFD2
.A 0375   RTS
```

Check it with a disassembly. If you disassemble starting with the subroutine, you'll need more than one screen full of instructions to see it all. No problem. When the cursor flashes at the bottom of the screen, press the letter D and RETURN and you'll see a continuation of the listing.

Back to BASIC. This time we do *not* give SYS 828—that's the subroutine and we want the main routine, remember?

Give the SYS 850 command. Tap a couple of numeric keys that total nine or less. Watch the results appear instantly on the screen.

If you like, set up a BASIC loop and call the routine several times.

Project for enthusiasts You couldn't resist, could you? You had to type in two digits that totaled over 9 and got a silly result OK, your project is to try to expand the above code to allow for two-digit results. It's not that hard, since the highest possible total is 9 + 9 or 18; so if there are two digits, the first one must be the digit 1. You'll need to compare for the result over binary nine, and then arrange for printing the one and subtracting ten if necessary. Sounds like fun

Things You Have Learned

—We may decide to use a number as a *signed* value; in this case, the high bit of the number will be 0 if the number is positive and 1 if the number is negative It's up to us As far as the computer is concerned, it's just bits in either case

—When a number might have a value that won't fit into an eight-bit byte, we may use more than one byte to hold the value. We have already done this to hold addresses in two bytes there's a *high byte* to hold the high part of the value and a *low byte* to hold the low part

—We may add two numbers together using the ADC instruction with the A register; we should always clear the carry flag before starting an addition. The carry flag will take care of multibyte numbers for us. providing we remember to start the addition at the low end

—We may subtract two numbers using the SBC instruction with the A register, we should always set the carry flag before starting a subtraction. The carry—which is sometimes called an *inverted borrow*—will take care of multibyte numbers for us, providing we remember to start the subtraction at the low end

—For unsigned numbers, the carry should end up as it started (clear for addition, set for subtraction), otherwise we have overflow in the result For signed numbers, the carry doesn't matter, the V flag will be set if we have overflow.

—We may multiply a byte by two with the ASL (arithmetic shift left) instruction If we have a multiple-byte number, we may carry the multiplication through to other bytes by using the ROL (rotate left) instruction, starting at the low byte of the number

—We may divide a byte by two with the LSR (logical shift right) instruction. If we have a multiple-byte number, we may carry the division through to other bytes by using the ROR (rotate right) instruction, starting at the high byte of the number

—The shift and rotate instructions may be used on the contents of the A register or directly on memory. The N and Z flags are affected, and the C flag plays an important role in the shift/rotate action.

—If we wish to multiply by a value other than two, we may need to do more work but we can get there

—As we might have expected, we may write subroutines in machine language and then call them from machine language It's a good way to organize your code.

Questions and Projects

Write a program to subtract two single-digit numbers, similar to the one in the above exercise. You may continue to use the subroutine from the previous chapter

Write a program to input a single-digit number. If the number is less than five, double it and print the result. If the number is five or over, divide it by two (discarding any remainder) and print the result. Try to produce a neat output.

Write a program to input a single-digit number. Print the word ODD or EVEN behind the number, depending on whether it is odd or even. Use the LSR instruction followed by a BCC or BCS test to check for odd or even

If you've been following the logic, you have developed quite a bit of capability in machine language. You can input, you can output, and you can do quite a bit of arithmetic in between.

By now, you should have developed skills with the machine language monitor and feel much more comfortable zipping in and out. These skills are not difficult, but they are important to the beginner. Without them, you can never get comfortably into the real meat: how to code machine language itself.
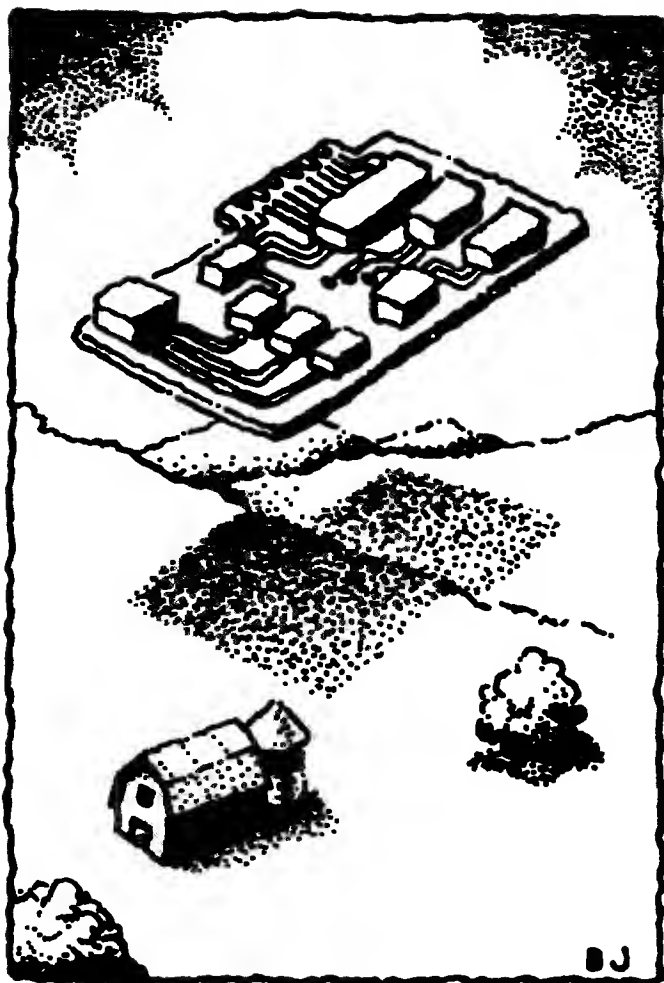
5

Address
Modes

This chapter discusses:

- Non-addresses: implied, immediate, register
- Absolute and zero-page
- Indexing
- The relative address for branches
- Indirect addressing
- Indirect, indexed

Addressing Modes

Computer instructions come in two parts: the instruction itself, or *op code*, and the address, or *operand*. The term "address" is a little misleading, since sometimes the operand does not refer to any memory address.

The term *address mode* refers to the way in which the instruction obtains information. Depending on how you count them, there are up to 13 address modes used by the 650x microprocessor. They may be summarized as follows:

1. No memory address. *implied, accumulator*
2. No address, but a value supplied. *immediate.*
3. An address designating a single memory location. *absolute; zero-page.*
4. An indexed address designating a range of 256 locations. *absolute,x; absolute.y; zero-page,x, zero-page,y*
5. A location in which the real (two-byte) jump address may be found: *indirect*
6. An offset value (e.g., forward 9, back 17) used for branch instructions. *relative*
7. Combination of indirect and indexed addresses, useful for reaching data anywhere in memory: *indirect, indexed; indexed, indirect*

No Address: Implied Mode

Instructions such as INX (increment X), BRK (break), and TAY (transfer A to Y) need no address; they make no memory reference and are complete in themselves. Such instructions occupy one byte of memory.

We might say that such instructions have "no address." The precise term is "implied address," which seems to say that there is in fact an address but we do not need to state it.

Perhaps the word "implied" is used in this manner: an instruction such as INX implies the use of the address register; and an instruction such as BRK implies the address of the machine language monitor. If so, there's an instruction that still defies this definition: NOP.

The Do-Nothing Instruction: NOP

NOP (no operation) is an instruction that does nothing. It affects no data registers or flags. When a NOP instruction is given, nothing happens and the processor continues to the next instruction. It seems inappropriate to

me that we say that NOP has an implied address. It doesn't do anything; it doesn't have an address at all On the other hand, I suppose that logicians might say, "Yes, but it does nothing to the X register."

The NOP instruction, whose op code is $EA, is surprisingly useful. It's not simply that if you're a contract programmer getting paid by the byte you might be tempted to put a large number of NOP instructions into your program. NOP can serve two important program testing functions: taking out unwanted instructions, or leaving space for extra instructions.

It's not as easy to change a machine language program as it is to change a BASIC program. As you have seen, the instructions are placed in specific locations. If we wish to eliminate an instruction, we must either move all the following instructions down or fill in the space with NOP instructions If we move the instructions, we may need to correct some of the addresses.

Examine the following code:

```
0350    LDA #$00
0352    STA $1234
0355    ORA $3456
```

If we decide to eliminate the instruction at 0352 (STA $1234), we must remove all three bytes. So we place code $EA in locations 0352, 0353, and 0354

Suppose we are testing a moderately large program Most programs will break into distinct "modules," each of which does a specific job. One module might clear a portion of memory to zero, another might do a calculation, and so on. When we are checking out this program, it might be wise to look at each module as it runs

In this case, we might deliberately code a BRK (break) command between each program module. The program will start to run, and then it will break to the machine language monitor. Within the monitor, we can examine memory to ensure that this module has done the job as we planned it. When we are satisfied, we can start the next module using the .G command. In this way, we can have tight testing control over our program.

That's all very well, but when we have finished testing our program and are satisfied that it runs correctly, we don't want the BRK instructions there That's easy to fix. We replace the BRK codes ($00) with NOP's ($EA), and the program will run through to the end

If we are writing a program and suspect that we may need to insert one or two extra instructions within a certain area of the code, we can put a

number of NOP instructions there The space will be available for use when we need it.

No Address: Accumulator Mode

We have observed that the shift and rotate instructions, ASL, ROL, LSR, and ROR, allow data manipulation in either the A register or directly in memory. When we want to use the A register, or *accumulator*, you should note this fact as you code your program. For example, you would write ASL A.

Where accumulator mode addressing is used, it has the same character- istics as implied addressing: the whole instruction fits into one byte.

Where the shift/rotate instruction refers to a memory location, an address will of course be needed. These address modes will be described later.

Other than the shift and rotate instructions, there is one other set of in- structions that manipulates memory directly. You may recall INX, INY, DEX, and DEY increment or decrement an index register.

INC (increment memory) adds one to any memory location. DEC (dec- rement memory) subtracts one from any memory location. Both instruc- tions affect the Z and N flags.

When an instruction modifies memory, the address mode is neither implied nor accumulator. Memory reference addressing will be discussed later.

Not Quite an Address: Immediate Mode

Coding such as LDA #$34 does not reference a memory address. In- stead, it designates a specific value (in this case, $34). An instruction with immediate addressing takes up two bytes: one for the op code and the second for the immediate value.

We have used immediate addressing several times It has a "natural" feel, and it's fast and convenient. There is one potential pitfall: immediate ad- dressing is so easy to use that it may be abused. Each time you code an immediate address, ask yourself, "Could this value ever change?" By writing a value into a program, rather than a variable, you may be freezing that value forever

An example: a program is written for a VIC-20, which has 22 columns on the screen. At various places in the program, values are compared to 22 (hex 16), and 22 is added or subtracted to various screen addresses. In

each case, immediate mode addressing is used to provide the value of 22 Some time later, the programmer decides to convert to the Commodore 64, which has 40 columns on the screen. The programmer must change each immediate mode reference from 22 to 40 (hex 28).

If the value 22 had been stored in a memory location so as to be used as a variable, all this recoding would not be needed. The moral is clear: excessive use of immediate mode can call for extra programming work at a later time.

There are certain instructions for which immediate addressing is not possible For example, we can LDA #$00, that is, bring in the actual value zero rather than the contents of an address, but we cannot STA immediate—we must store the information somewhere in memory.

A Single Address: Absolute Mode

An instruction might specify any address within memory—from $00000 to $FFFFF—and handle information from that address. Giving the full address is called *absolute addressing*; if you like, you can deal with information absolutely anywhere in memory.
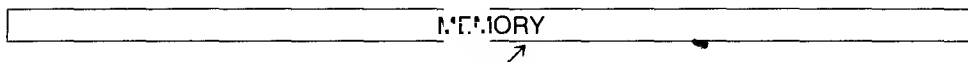
Figure 5.1 Absolute Mode Specifies One Address Anywhere Within Memory.

We have used absolute addresses several times. When we exchanged the contents of memory locations $0380 and $0381, we named these addresses as we used them. When we stored a value from the keyboard, we named location $03C0. We have also used absolute addresses for *program control*: subroutines at $FFD2 and $033C were called up simply by giving the address

The JSR (jump subroutine) instruction calls up a subroutine anywhere in memory by using absolute addressing. There is also a JMP (jump) instruction, which can transfer program execution to any location in memory; it's similar to the BASIC GOTO statement. JMP can use absolute addressing—it can go anywhere.

There's a limitation to absolute addressing, however. Once you have written the instruction, you can go only to the address stated. You cannot reach a range of locations; only one.

One-location addressing can be good for any of several jobs. On the PET/

CBM, we might want to switch between text and graphics modes by manipulating address 59468 (hexadecimal E84C). On the VIC-20, we might like to set the volume level of the sound generator by placing a value into location 36878 (hex 900E). On a Commodore 64, the screen's background color can be changed by manipulating address 53281 (hex D021). In each case, it's one specific address that we want; absolute addressing will do the job for us. And we will also use absolute addressing to reference the various RAM locations that we have picked for our own program "variables."

Zero-Page Mode

A hexadecimal address such as $0381 is sixteen bits long and takes up two bytes of memory. We call the high byte (in this case, $03), the "memory page" of the address. We might say (but usually don't) that this address is in page 3 at position $81.

$00 $FF $100

Figure 5.2 Zero-Page Mode Specifies A Single Address from $00 to $FF

Addresses such as $004C and $00F7 are in page zero; in fact, page zero consists of all addresses from $0000 to $00FF. Page-zero locations are very popular and quite busy. There's an address mode specially designed to quickly get to these locations: zero-page addressing. We may think of it as a short address, and omit the first two digits. Instead of coding LDA $0090, we may write LDA $90, and the resulting code will occupy less space and run slightly faster.

Zero-page locations are so popular that we'll have a hard time finding spare locations for our own programs. As a result, we tend to conserve zero-page locations on Commodore machines. We'll need the few that are available for a special addressing mode, *indirect, indexed*, that will be discussed later.

There are many locations in zero page that are useful to read. For example, the BASIC system variable ST, which is important in input/output handling, may be examined there (location $96 in PET/CBM, location $90 in VIC-20 and Commodore 64). If you need to know whether the user is holding down a key, there's an address in zero page that will tell you that (location $97 in PET/CBM, $CB in VIC and 64).

Zero-page addressing, like absolute addressing, references one location

only. It's good for a specific value; but for a range of values we need something more.

A Range of ੨5੬ Addresses: Absolute, Indexed Mode

Indexing has already been used in Chapter 2. We give an absolute address, and then indicate that the contents of X or Y should be added to this address to give an *effective address*.
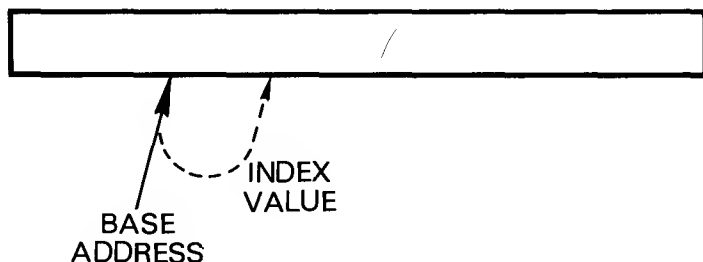
Figure 5.3

Indexing is used only for data handling: it's available for such activities as load and store, but not for branch or jump. Many instructions give you a choice of X or Y as an index register; a few are limited specifically to X or Y. Instructions that compare or store X and Y (CPX, CPY, STX, and STY) do not have absolute, indexed addressing; neither does the BIT instruction.

An instruction using absolute, indexed addressing can reach up to ੨5੬ locations. Registers X and Y may hold values from ੦ to ੨55, so that the effective address may range from the address given to ੨55 locations higher. Indexing always *increases* the address; there is no such thing as a negative index when used with an absolute address. If the address given is above $FF੦੦, a high value in the index may cause the address to "wrap around" and generate an effective address in the region of $੦੦੦੦; otherwise, the effective address is never lower than the instruction address.

We've seen the use of indexing. An instruction can reference a certain address, then, as the program loops or as the need for information changes, the same instruction can reference the contents of a different address. The maximum range of ੨5੬ locations is an important limitation.

The "reach" of an absolute, indexed instruction allows it to handle information in buffers (such as the input buffer, keyboard buffer, cassette buffer); tables (such as the active file table); and short messages (such as HELLO or error messages). It's not big enough, however, to reach all parts of screen memory, all parts of a BASIC program, or all of RAM. For that, we'll use *indirect, indexed* addressing, which will be described later.

All of Zero Page: Zero-Page, Indexed

Zero-page, indexed addressing seems at first glance to be similar to the absolute, indexed mode The address given (this time in zero-page) has the contents of the selected index added to it. But there's a difference: in this case, the effective address can never leave zero page

This mode usually uses the X register, only two instructions, LDX and STX, use the Y register for zero-page, indexed addressing In either case, the index is added to the zero-page address; if the total goes beyond zero page, the address "wraps around" As an example, if an instruction is coded LDA $EO,X and the X register contains 5O at the time of execution, the effective address will be $OO3O. The total ($EO + $5O or $13O) will be trimmed back into zero page

$OO $FF

BASE
ADDRESS
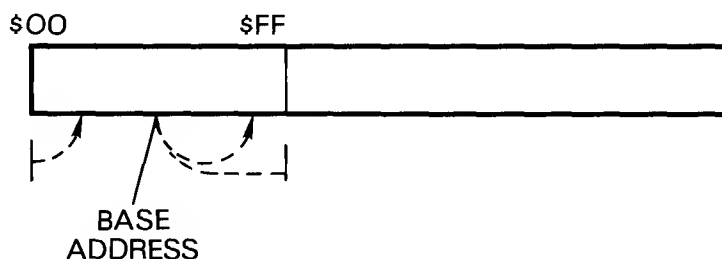
Figure 5.4

Thus, any zero-page address can be indexed to reach any other place in zero page; the reach of 256 locations represents the whole of zero page. This creates a new possibility: with zero-page, indexed addressing, we can achieve negative indexing For this address mode only, we can index in a downward direction by using index register values such as $FF for −1, $FE for −2, and so on

On Commodore machines, zero page is fairly well occupied. There is limited opportunity to use zero-page, indexed addressing.

Branching: Relative Address Mode

We have written several branch instructions already, the assembler allowed us to enter the actual addresses to which we want to branch The assembler translates it to a different form—the relative address
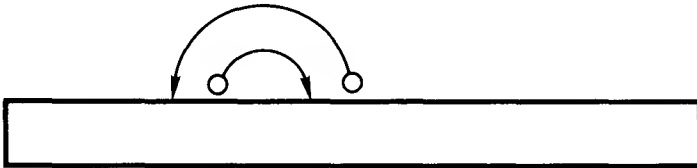
Figure 5.5

Relative address means, "branch forward or backwards a certain number of bytes from this point " The relative address is one byte, making the whole instruction two bytes long. Its value is taken as a signed number.

A branch instruction with a relative address of $ 0 5 would mean, "if the branch is taken, skip the next 5 bytes." A branch instruction with a relative address of $ F 7 would mean, "if the branch is taken, back up 9 bytes from where you would otherwise be." As a signed number, $ F 7 is equal to a value of − 9

We can calculate a branch by performing hexadecimal subtraction; the "target" address is subtracted from the P C address If we have a branch at $ 0 3 4 1 that should go to $ 0 3 3 C, we would work as follows: $ 0 3 3 C (the target) minus $ 0 3 4 3 (the location following the branch instruction) would give a result of $ F 9, or minus 7. This is tedious to do, and often results in mistakes; such mistakes in calculating a branch address are often fatal to the program run. We are much better off using an assembler to work out the arithmetic for us.

The longest branches are $ 7 F, or 1 2 7 locations ahead; and $ 8 0, or 1 2 8 locations back. This poses no difficulties with short programs, such as the ones we are writing here. But in larger programs, the branch may not be able to reach far enough The usual solution to this is to place a J M P (jump) instruction nearby, which is capable of going anywhere in memory; J M P uses absolute addressing. The appropriate branch instruction will go to the J M P, which in turn will take the program to the desired location.

Advocates of programming style make the following argument. All programs should be written into neat small modules. Logic blocks should be

broken into subroutines, and the subroutines into even smaller subroutines; this way, everything is neat and testable. If you should find a branch that won't reach, ask yourself whether it's time to break your program into smaller chunks before the logic gets too messy By the liberal use of subroutines, you can arrange your code so that all branches are short and easily within reach If you do break up the program structure, the branches will then always reach It's up to you to choose your coding style, but you might give the question some thought.

An interesting aspect of relative addressing is that code containing branches is easy to relocate. A piece of code containing a branch to six locations ahead will work perfectly if the whole code is moved to a different location. This is not true of jumps and subroutine calls, or any code using absolute addressing—if the location changes, the address must be changed.

The ROM Link—Jumps in Indirect Mode

We have mentioned the JMP instruction that will take the program to any specified address. JMP has another address mode: *indirect addressing.*

Indirect addressing is signaled by the use of parentheses around the address. It works this way. An address is supplied, but it's not the one we will eventually use. We take this address, and at the location it specifies, we'll find the effective address, or the *indirect address.* The indirect address is two bytes long, of course, and is stored in the usual 650x manner of low byte first.

An example will help to make things clear. Suppose that at address $033C we have the instruction JMP ($1234). The parentheses tell us that indirect addressing is involved. The machine code is hex 6C 34 12; as always, the address is "turned around." Now suppose that at addresses $1234 and $1235 we have stored values $24 and $68. The jump instruction would behave as follows: it would go to $1234 and $1235, get the contents, and the program would transfer to address $6824.
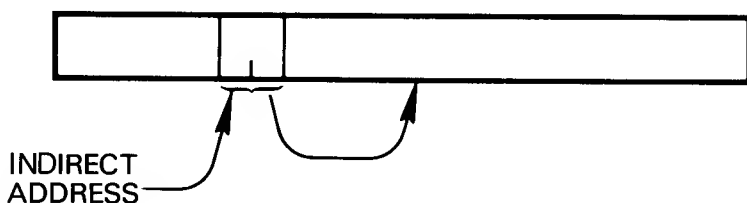

INDIRECT
ADDRESS

Figure 5.6

The JMP indirect has a somewhat specialized use. Normally, if we want to transfer control to some location, we just JMP there; no need for the indirect step. But there's one quite important case where indirect jumps serve an important function.

Within ROM, there are a large amount of permanent instructions that the computer uses to perform its tasks. Since it's in ROM, we can never change this code. If the various programs were linked only by means of JMP and JSR statements, they could not be changed, and we would not be able to modify the behavior of the machine.

Built into the ROM program, there are a series of carefully planned indirect jumps. Instead of the ROM leaping from one instruction directly to another, it jumps indirectly via an address stored in RAM. We can change the contents of RAM; and if we change the address stored in RAM, we can modify the behavior of the system. The best-known indirect address is that associated with the interrupt sequence it's at $0090 in PET/CBM and $0314 in VIC, 64, and PLUS/4.

You might not code many indirect jumps, but you'll be glad that they are there in ROM.

Data From Anywhere: Indirect, Indexed

The problems with indexed addressing have been noted the reach of only 256 bytes limits the data capability of this method

Indirect addressing seems to offer a total solution. We can write an instruction that points at an indirect address. Since we can change the indirect address at will, or add to or subtract from it, we can cause our instruction to deal with data anywhere in memory.

In fact, we get a limitation and a bonus. First, the limitation for indirect, indexed instructions the indirect address must be in zero-page—two bytes, of course, organized low byte first, as always. Next, the bonus: after the indirect address is obtained, it will be indexed with the Y register to form the final effective address.

Let's step our way through the mechanism and see how it works. Suppose I code LDA ($C0), Y with values $11 in address $00C0 and $22 in address $00C1. If the Y register contains a value of 3, the instruction will follow these steps: The address at $00C0-1 is extracted, giving $2211; then the contents of Y are added to give the effective address of $2214. If the contents of Y changed, the effective address would

change slightly. If the indirect address at $C0 and $C1 was changed, the effective address would change radically.

The combination of indirect and indexing may seem like overkill. If you can designate any location in memory with an indirect address, why bother with indexing? After all, anywhere plus one is still anywhere.

Indirect addressing plus indexing proves to be an ideal combination for the manipulation of data. Almost all data breaks up into logical chunks of some sort: records, table entries, screen lines, words, and so on. Here's the technique. We position the indirect address at the start of a given logical data chunk, and use the Y register to scan through the information. When we're ready to move to the next item, we move the indirect address along, and repeat the same scanning of the Y register through the new data.
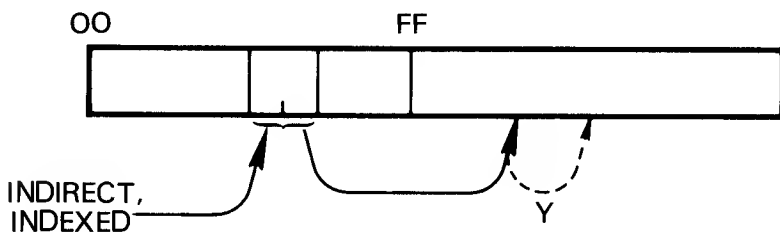
Figure 5.7

One may think of it as a fishing analogy: We anchor the boat in a certain spot (fix the indirect address) and then use the fishing line (the Y register) to reach the data we need. When we're ready for the next item, we pull up the anchor and move along to a new place.
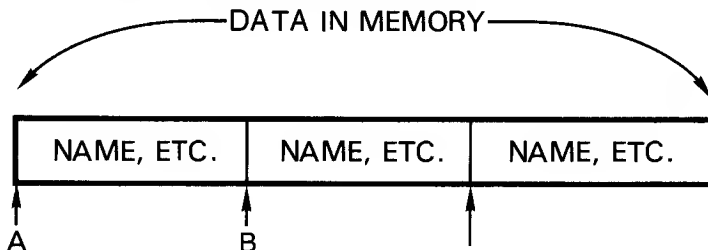
Figure 5.8

We'll be working through an elaborate example that uses indirect, indexed addressing to manipulate the computer screen. First, a brief diversion.

A Rarity: Indexed, Indirect

There is another addressing mode that is little used in Commodore computers· indexed, indirect. It uses the X register rather than the Y, and is coded as in the following example: LDA ($C0,X). In this case, indexing takes place first. The contents of X are added to the indirect address (in this case, $C0) to make an *effective indirect address*. If X were equal to 4 in this example, the effective indirect address would be $00C4, and the contents of $00C4 and $00C5 would be used as the effective address of the data
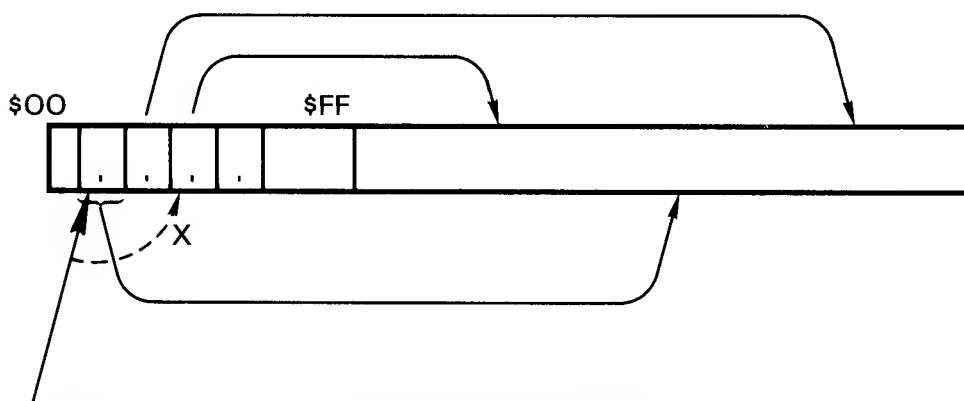
INDEXED, INDIRECT ALLOWS ONE OF SEVERAL
INDIRECT ADDRESSES TO BE CHOSEN USING
THE X INDEX REGISTER

Figure 5.9

In certain types of control processing, this is a quite useful address mode. X will contain an even number; since each indirect address is two bytes long, we will need to skip from one to the other, two bytes at a time.

Let's take a hypothetical communications system that is connected to four telecommunications lines and see how indexed, indirect addressing might be used. Characters are being received from the four lines almost simultaneously. As each character arrives, it must be put away into a memory buffer belonging to that particular line; in that way, traffic received from the various sources won't get mixed together. Zero-page will contain four indirect addresses, one for each line; each indirect address points at an input area for one line. Suppose a character is received into the A register from one of the lines, the line number (times two) is in the X register. We

could then put the character away with the instruction STA ($60,X). Thus, if line zero was involved, its indirect address at address $60/61 would be used; for line 1, the address at $62/63 would be used: and so on. After we had stored the character concerned, we'd need to bump the indirect pointer so that the next character will go into a new position: INC $60,X would do the trick.

The above example is a rather specialized use of the indexed, indirect address mode. You may never need to use this mode. Indeed, most programmers lead full, rich lives without ever writing code that uses indexed, indirect addressing.

The Great Zero-Page Hunt

Indirect, indexed addresses are very important. They are your gateway to reaching any part of memory from a single instruction. But you must have two bytes available in zero-page for each indirect address you want to use.

The Commodore ROM system helps itself to liberal amounts of zero-page memory You don't have much empty space left over. How can you find space for these indirect pointers?

First, look for unused locations. There are only a few of them: on the VIC and Commodore 64, you'll find four locations at locations $00FC to $00FF. That's enough for two indirect addresses

If you need more, look through the memory maps for locations designed as "work areas" or "utility pointers." They can usually be put to work for a temporary job

Finally, you can take working parts of zero-page and copy them to some other parts of memory You can use these locations, carefully putting back the original contents before returning to BASIC. Don't try this with any values that are used by the interrupt routines (involved with screen, keyboard, or RS-232); the interrupt can and does strike while your machine language program is running And if the interrupt program changes these zero-page values, your program is going to behave badly.

Project: Screen Manipulation

This project is intended to show how indirect, indexed addressing can be used effectively. We'll change something on the screen—enough so that we reach more than 256 addresses. Ordinary indexing, therefore, won't do.

We'll select a number of lines on the screen; within each line, we'll change a certain group of characters. In other words, we will write the code so as to manipulate a *window* on the screen

To do this, we'll need to code two steps: setting up the start of a screen line, and later moving on to the next line when needed. Within each line, we'll work our way through the range of screen columns that we have selected. In fact, it's a big loop (for the lines) containing a small loop (for the columns within that line). We'll use indirect addressing to point to the start of each line, and indexing (the Y register) to select the portion of that line to change.

Since there's a variety of Commodore machines, we have some problems to resolve. All Commodore screens are "memory mapped," that is, the information appearing on the screen is copied directly from some part of memory. We may change the screen by changing the appropriate memory But different machines use different memory addresses; and in VIC and Commodore 64, the screen may be moved around. Another thing to consider is that the length of line varies between different machines—it might be 22 or 40 or 80 columns.

No problem. If you have a 40-column machine, 40 equals $28; code

```
.A 033C LDA #$28
```

For a 22-column machine, change the above to LDA #$16; and for an 80-column PET, code LDA #$50.

Have you coded the correct value? Let's proceed with our next decision. In the PET/CBM, screen memory starts at address $8000; in VIC or Commodore 64, the screen starts at whatever page is designated in address $0288. Let's code as follows:

```
PET/CBM:              .A 033E LDX #$80
                      .A 0340 NOP
VIC/Commodore 64:     .A 033E LDX $0288
```

The NOP instruction does nothing, but it makes the coding the same length so that we may continue with address $0341 in either case. The A register tells us our line length, and the X register tells us the page number on which the screen starts. Let's put them away. The line length will be needed for addition later, so we may put it anywhere safe; the screen address will be part of an indirect address, so it must go into zero-page.

It's hard to find a zero-page address that may be used in all Commodore

machines; we'll choose $00BB and $00BC. $BB contains the low byte of the address, of course. Let's code

```
.A 0341 STA $03A0
.A 0344 STX $BC
```

Note that we are using the zero-page addressing mode for the instruction at address $0344. That puts the high byte of the address in place. Now we'll set the low byte to zero:

```
.A 0346 LDA #$00
.A 0348 STA $BB
```

Our indirect address is now pointing at the start of screen memory. Let's discuss in more detail what we want to do with the screen. Specifically, we want to change a number of lines, let's say 14, on the screen. We will step along our indirect address by adding to the indirect address: maybe 22, maybe 40, maybe 80; whatever is in address $03A0. And we won't do the whole line; we'll start in column 5 and go to column 18. Let's count the lines in the X register; we'll start X at zero

```
.A 034A LDX #$00
```

Now we're ready to do a screen line. Later, we'll adjust the indirect address and come back here to do another line. We should make a note to ourselves: "Come back to $034C for the next screen line."

The indirect address is pointing at the start of the line. We want to start work in column 5. That means that Y should start with an offset of 4 (the start of the line plus 4). Let's do it:

```
.A 034C LDY #$04
```

We're going to walk Y up, and loop back to this point for the next character on the line. We might note: "Come back to $034E for the next character."

We're ready to go. Let's dig out the character that's currently on the screen:

```
.A 034E LDA ($BB),Y
```

This is worth a review. Locations $BB and $BC contain the address of the start of screen memory; on the PET/CBM, for example, this would be $8000. To this, we add the contents of Y (value 4) to create an effective address of $8004; and from location $8004 we get the screen character.

We decide that we will leave spaces alone. The space character shows on the screen as a value of decimal 32, hex 20. Let's skip the next operation if it's a space:

```
.A 0350 CMP #$20
.A 0352 BEQ $0356
```

We have to guess at the address to which we will skip ahead, since we haven't gotten there yet. Make a note: "This address may need correction."

```
.A 0354 EOR #$80
```

This is where we manipulate the character. The EOR is a "flip-over" command; we're flipping the high bit of the screen value. You may look up screen codes to see what this does, or you may wait and see what happens. At this point, our code from $0352 joins up. As it happens, we were lucky again: the address is exactly right to rejoin at $0356. But if it were not, you know how to fix it, don't you? Exit the assembler, then go back and type over.

Now we put the modified character back to the screen:

```
.A 0356 STA ($BB),Y
```

We have done one character. Let's move along the line to the next character, and if we have passed column 18 (Y = 17) we should quit and go to the next line.

```
.A 0358   INY
.A 0359   CPY #$12
.a 035B   BCC $034E
```

Y moves along to the next character position: five, then six the next time around, and so on. So long as Y is less than 18 (hex 12) we'll go back, since BCC means "branch less than." If we get past this point, we have completed the line and must move to the next one.

We move to the next line by adding to the indirect address. We must add 22, or 40, or 80; the value is in address $03A0 (you may remember that we stored it with the instruction at $0341). We must remember to clear the carry flag before starting the addition, and to add starting at the low byte of the address (at $BB).

```
.A 035D   CLC
.A 035E   LDA $BB
.A 0360   ADC $03A0
.A 0363   STA $BB
.A 0365   LDA $BC
.A 0367   ADC #$00
.A 0369   STA $BC
```

The last three instructions seem odd. Why would we add zero to the contents of $BC? Surely that changes nothing. The answer is obvious after a little thought: there might be a carry from the previous addition.

Now we're ready to count the lines: we had decided to use X as a counter. Let's add one to X, and test to see whether we have done the 14 lines:

```
.A 036B INX
.A 036C CPX #$0E
.A 036E BNE $034C
```

If we've done the required number of lines, we have nothing more to do other than return to BASIC:

```
.A 0370 RTS
```

Disassemble and check it. Again, you'll find that the code occupies more than one full screen. Return to BASIC.

This time, we'll write a small BASIC program to exercise the machine language code. Type NEW to clear out any old BASIC code, and enter

```
100 FOR J = 1 to 10
110 SYS 828
120 FOR K = 1 to 200
130 NEXT K,J
```

The extra loop is to slow things down. Machine language runs so fast that the effect might not be properly visible if run at full speed.

Project for enthusiasts: Can you change the program to do a different set of columns? Could you change it so that it affected only the letter "S" wherever it appeared on the screen?

Comment for VIC-20 and Commodore 64

This exercise will work as intended. Other types of screen work might call for you to set the color nybble memory values before you can successfully work directly with screen memory. The rules for machine language are no different from those for BASIC: if you wish to POKE to the screen, you may need to take the color nybble area into account.

Things You Have Learned

—Three *address modes* are not addresses at all. *Implied* addressing means no address at all, *accumulator* addressing uses the A register and means the same thing, and *immediate* addressing uses a value, not an address

—*Absolute* addresses reference one location only, somewhere in memory. *Zero-page* addresses reference a single address in the range $0000 to $00FF—the high byte of the address (00) is the *memory page*. These address modes are used for fixed locations containing work values or system interfaces.

—*Absolute, indexed* and *zero-page, indexed* allows the named address to be adjusted by the contents of an *index register*—X or Y. These instructions can reach a range of up to 256 addresses. They are commonly used for tables of data or temporary storage areas.

—*Relative* addresses are used exclusively with branch instructions. They have a limited "reach" of about 127 locations forward or backward. It takes a little arithmetic to calculate the proper values, but the computer usually works this out for us.

—*Indirect* addressing is used only for jumps, most often to allow a fixed ROM program to take a variable jump. The average machine language programmer will seldom need these, but the principle of indirect addressing is worth learning.

—*Indirect, indexed* addressing is the most important way to deal with data anywhere in memory. We may reach anywhere by setting the indirect address, then we may "fine adjust" that address by indexing it with the contents of Y.

—Indirect, indexed addressing requires the indirect address to be in zero-page. We need to conserve zero-page locations for this use.

—An addressing mode called *indexed, indirect* is rarely used when programming Commodore computers, but it's there if you want it.

Questions and Projects

Write a program to clear the screen of your computer—check Appendix C for the location of screen memory if you've forgotten. Don't just print the clear screen character ($93); do it another way. Can you write the entire program without using indirect, indexed addressing?

Write the program again using indirect, indexed addressing. The program may be a little shorter. Can you think of any other advantages of writing this way?

A user wishes to type in a line of text on the keyboard, ending with a RETURN. He then wants to have the program repeat the line ten times on the screen. What addressing mode or modes would you use to handle the user's text? Why? You may try your hand at writing the program if you wish.

Take one of the previous exercises and try to write it again without using immediate addressing. Is it hard to do? Can you see any reason to want to code without using immediate addressing at all?
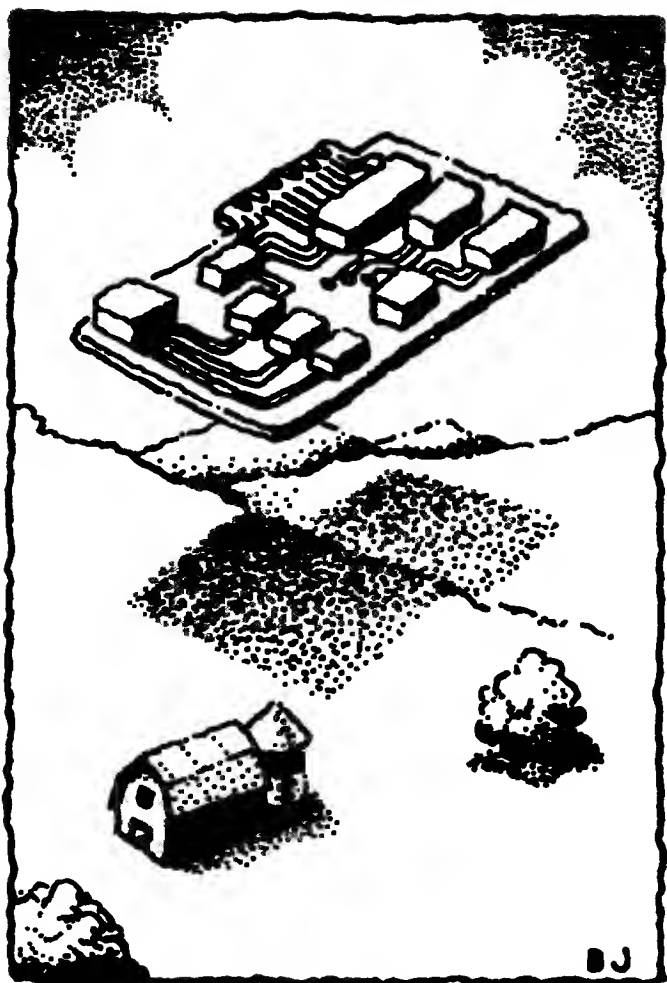
6

Linking BASIC and Machine Language

This chapter discusses:

- Where to put a machine language program
- BASIC memory layout
- Loading and the SOV pointer
- BASIC variables fixed, floating and string
- Exchanging data with BASIC

Siting the Program

Up to this point, we have been placing all programs in the cassette buffer. This is a good place for short test programs, but we need to examine alternatives that are often more attractive

BASIC Memory Layout

BASIC R A M is organized according to the diagram below. The following locations are of particular interest:

1. Below the BASIC area, we have the cassette buffer area This is available to us, providing we are not engaged in input/output activity

2. Start-of-BASIC (SOB) is usually a fixed address within the machine In PET/CBM, it's at $0401 (decimal 1025) In Commodore 64, it's at $0801 (decimal 2049) In the PLUS/4 series, it's at $1001 (decimal 4097) In the VIC-20, it may be at one of several places $0401, $1001, or $1201. A pointer marks this location. The pointer is located at $28/$29 (decimal 40 and 41) in PET/CBM, and at $2B/$2C (decimal 43 and 44), in VIC-20, Commodore 64, and PLUS/4.

   You should inspect the pointer and confirm that it contains an appropriate address. You may notice that it's much easier to do this using the machine language monitor, since the address is split between the two bytes (low order first, as always).

3. End-of-BASIC is signaled by three zero bytes somewhere after the SOB. If you command NEW in BASIC, you'll find the three bytes right at the start of BASIC, there is no program, so start and end are together. There is no pointer that indicates end-of-BASIC, just the three zeros; but the next location (SOV) will often be directly behind the end-of-BASIC

   The BASIC program that you type in will occupy memory space from start-of-BASIC to end-of-BASIC If you add lines to a program, end-of-BASIC will

BASIC RAM

CASSETTE BUFFER		Ø	BASIC PROGRAM	Ø Ø Ø	BASIC VARIABLES	BASIC ARRAYS	FREE	DYNAMIC STRINGS

SOB   SOV   SOA   EOA   BOS   TOM
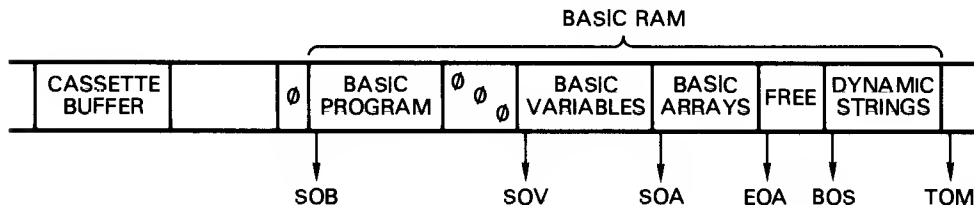
Figure 6.1

move up as extra memory is taken up by your programs. If you delete lines, end-of-BASIC will move down

4. Start-of-variables (SOV) is often positioned directly behind the end-of-BASIC. When the BASIC program runs, the variables will be written into memory starting at this point; each variable is exactly seven bytes long. A pointer marks this location. The pointer is located at $2A/$2B (decimal 42 and 43) in PET/CBM, and at $2D/$2E (decimal 45 and 46) in VIC-20, Commodore 64, and PLUS/4

The SOV pointer is extremely important during BASIC load and save activities. If we give the BASIC command SAVE in direct mode, the computer will automatically save all memory from SOB to just before the SOV. Thus, it saves the whole BASIC program, including the end-of-BASIC marker of three zero bytes, but does not save any variables. If we give the BASIC command LOAD in direct mode, the computer will automatically load the program, and then place the SOV pointer to just behind the last byte loaded. In this way, variables will never be stored over the BASIC program; they will be written above the end-of-BASIC. More on this later

If the BASIC program is changed, the SOV may move up or down as needed

5. Start-of-arrays (SOA) also represents one location beyond the end-of-BASIC variables, and thus could be named end-of-variables. Arrays created by the BASIC program, either by use of a DIM statement or by default dimensioning, will occupy memory starting at this point. A pointer marks this location. The pointer is located at $2C/$2D (decimal 44 and 45) in PET/CBM, and at $2F/$30 (decimal 47 and 48) in VIC-20, Commodore 64, and PLUS/4. If the BASIC program is changed, the SOA pointer is set to match the SOV. Thus, all BASIC variables are wiped out the moment a change is made to the program

6. End-of-arrays (EOA) is set one location beyond the last array location in BASIC. Above this point is seemingly "free" memory—but it's not really free, as we'll see soon. A pointer marks this location. The pointer is located at $2E/$2F (decimal 46 and 47) in PET/CBM, and at $31/$32 (decimal 49 and 50) in VIC-20, Commodore 64, and PLUS/4

If the BASIC program is changed, the EOA pointer is set to match the SOA and SOV. Thus, all BASIC arrays are wiped out the moment a change is made to the BASIC program

Let's change direction and start to work our way down from the top of BASIC memory.

7. Top-of-memory (TOM) is set one location beyond the last byte available to BASIC. On the PET/CBM and VIC-20, its location depends on the amount of memory fitted; a 32K PET would locate TOM at $8000. On the Commodore 64, the TOM will normally be located at $A000. A pointer marks this location. The pointer is located at $34/$35 (decimal 52 and

53) in PET/CBM, and at $37/$38 (decimal 55 and 56) in VIC-20, Commodore 64, and PLUS/4

If you examine the TOM pointer, you may find that it does not point at the expected position. That may be because of the machine language monitor, which has taken up residence at the top of memory and stolen away some memory

8. Bottom-of-strings, (BOS) is set to the last "dynamic" string that has been created. If there are no BASIC strings, the BOM will be set to the same address as TOM. As new dynamic strings are created, this pointer moves down from the top-of-memory towards the EOA address A pointer marks this location The pointer is located at $30/$31 (decimal 48 and 49) in PET/CBM, and at $33/$34 (decimal 51 and 52) in VIC-20, Commodore 64, and PLUS/4.

A *dynamic string* is one that cannot be used directly from the program where it is defined; you might like to think of it as a manufactured string. If, within a BASIC program, I type: 100 X$ = "HAPPY NEW YEAR", the BASIC interpreter will not need to store the string in upper memory; it will use the string directly from where it lies within the program. On the other hand, if I define strings with commands such as R$ = R$ + "*" or INPUT N$, the strings must be built into some spare part of memory. That's where the BOS pointer comes in: the computed string is placed high in memory, and the BOS moved down to mark the next free place.

If the BASIC program is changed, the BOS pointer is set to match the TOM. Thus, all strings are wiped out the moment a change is made to the BASIC program.

Free Memory: The Dangerous Place

It seems to beginners that there is a great deal of free memory available above the end-of-arrays and below the bottom-of-strings, and that this would be an ideal place to put a machine language program. This is a pitfall: it usually won't work.

Here's the danger. As more and more dynamic strings are created, the bottom-of-strings location keeps moving down. Even when strings are no longer needed, they are abandoned and left dead in memory, taking up space.

The BOS keeps moving down. Only when it touches the EOA will the dead strings be cleaned up and the good ones repacked, an action called *garbage collection*. It's important for BASIC programmers to know about garbage collection: except on BASIC 4.0 and Commodore PLUS/4 systems, it can be a cause of serious program slowdown.

It's evident that the space between EOA and BOS is not safe. If you put a program there, the strings will eventually destroy it. We must look elsewhere.

Where to Put Your ML Program

First, you may put your program in the cassette buffer. Providing you are not performing input/output activity, your program will be safe. Your space here is limited to 1٩0 characters or so.
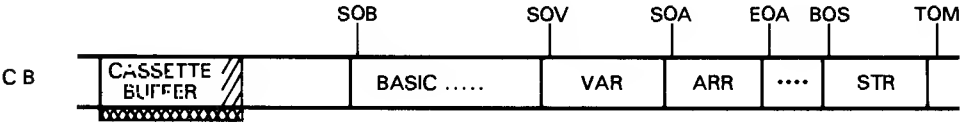
Figure 6.2

Second, move down the top-of-memory pointer and place the program in the space that has been freed. Your space here is unlimited. Programs placed here will take up permanent residence until the power is turned off. Many monitors, such as Supermon, live here.
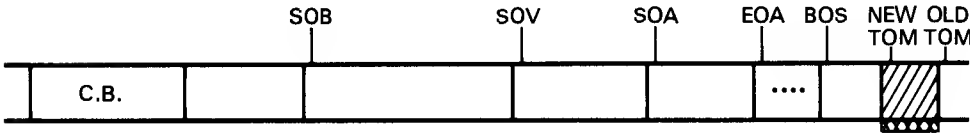
Figure 6.3
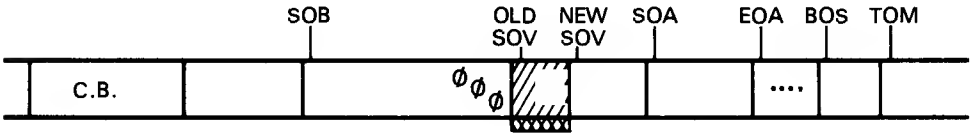
Figure 6.4

Third, move up the start-of-variables pointer, and place the program after the end of BASIC and before the new start-of-variables. Your space here is unlimited. Programs placed here will tend to "join company" with the BASIC program; the two will save and load together.

After moving a pointer—as was done in the last two methods—it's a good idea to return to BASIC and command CLR, so that all other variable pointers will align correctly with the ones that have moved.

These three areas will be discussed more in a few moments. First, there are one or two extra locations available to VIC-20 and Commodore 64.

Extras for VIC and Commodore 64

The Commodore 64 has a free block of RAM at locations $C000 to $CFFF (decimal 49152 to 53247) That's 4K of RAM not being used; you may write your programs there. Before you do so, check to make sure that the memory is not being used by any other programs It's a popular place in the Commodore 64, and many utilities and commercial programs zero in on this available memory.

If you intend to write programs entirely in machine language, with no BASIC content at all, you may completely remove BASIC from the Commodore 64 system and claim the space as available RAM. This gives you the whole block from $0801 up to $CFFF for programs and data—a whopping 94K—and even more could be liberated if necessary. BASIC may be made to disappear from the Commodore 64 with the equivalent of POKE 1,54 (LDA #$36, STA $01). It may be reinstated with the equivalent of POKE 1, 55 (LDA #$37, STA$01). Be very careful. With BASIC gone, the computer doesn't even know how to say READY.

On all Commodore machines it's possible to move up the start-of-BASIC pointer and use the space freed below. To do so, it's essential to store a value of zero into the location immediately before the new start-of-BASIC, and to align all other pointers, usually by going to BASIC and commanding NEW.

This works, and will make as much space available as is needed. BASIC programs will relocate as they load. But since the computer needs to be reconfigured before the main program is loaded, and often needs to be restored to its original configuration after the program is run, the method is not popular in most Commodore machines. It's used fairly often in the VIC-20, however.

The video chip in the VIC-20 can "see" RAM memory only in the memory space $0000 to $1FFF (decimal 0 to 8191). Whatever variable information appears on the screen must be taken from this memory area. The VIC-20 can also get information from $8000 to $9FFF, but there's no RAM there; we can't manipulate this memory area.

If we want to perform special visual effects on the VIC-20, we must manipulate data in the area $0000 to $1FFF. Let's look at what is available. $0000 to $03FF is used by the "system;" other than the cassette buffer, we must leave it alone. $0400 to $0FFF contains no memory unless a 3K RAM expansion is added. $1000 to $1DFF contains the BASIC program, and $1E00 to $1FFF is screen memory. Details may vary, but the answer always comes out the same: there's no space to do our video effects.

A popular VIC-20 solution, especially where 8K or more of RAM expansion has been added, is to increase the start-of-BASIC pointer, thus liberating space in low memory. This may now be used for visual effects and for machine language programming, too, if any space is left over. In the VIC-20, this approach is necessary, but it's still a bit clumsy.

The Wicked SOV

The start-of-variables pointer can be the cause of many troubles, if it's not understood. The rules covering it are as follows:

1. Variables are written starting at the SOV.

2. BASIC SAVEs will save from memory beginning at start-of-BASIC and stopping at SOV.

3. Direct command BASIC LOADs will bring a program into memory, relocating if appropriate, and then set the SOV pointer to the location following the last byte loaded.

4. Changes to BASIC programs cause memory to be moved—up or down—starting from the point where the change is made and stopping at the SOV. The SOV will then be moved the appropriate distance up or down.

These seem to be innocent rules. Rule 1 defines the purpose of the SOV. Rule 2 shows how the SOV controls the SAVE command so that the entire BASIC program is saved, but not the variables. Rule 3 arranges that short programs will have a large amount of variable space available; long ones will have less. Rule 4 ensures that a BASIC change makes extra room in memory or reclaims memory space.

But if the SOV gets the wrong address, we're in trouble. The rules work against us. Variables may be written into disastrous places. SAVEs will cause too much or too little to be saved. LOADs may fix things, since SOV will be changed by the load action. An attempt to change a program with a bad SOV may cause too little or far too much memory to be moved around. We must get the SOV right.

How can the SOV go bad on us? Let's take three examples, corresponding to the three major places that we might put machine language programs:

We have a program in the cassette buffer, and a BASIC program that goes with it. We enter or load the BASIC program (the SOV is all right so far), and then we $LOAD$ the machine language program; the SOV ends up disastrously somewhere in the cassette buffer area.

We're in trouble. The program seems to list correctly, but it's sick. If we RUN, variables will start to be placed in the cassette buffer area; as more variables are created, they are placed in progressively higher memory locations. Eventually, the variables start to write over the BASIC program. Everything stops. The poor programmer says $LIST$ to see what's happened; his BASIC program is gone, and all that's left is gibberish.

We're in more trouble. Alternatively, the programmer decides to save his BASIC program and commands $SAVE$. BASIC starts to save memory beginning at start-of-BASIC . . . and keeps saving, and saving, and saving. It won't stop until it reaches the SOV, but that's below where we started. We won't get there until the address "wraps around" and comes back up through zero. The poor programmer—if he or she waits long enough—discovers that the tiny five-line BASIC program has been saved as over 250 blocks on disk, or fifteen minutes worth of tape. And the saved program is useless.

We're in still more trouble. Alternatively, the programmer lists the program, and decides to delete one character from a line of BASIC. BASIC immediately starts to move memory, starting at the change point. It won't stop moving memory until it reaches SOV, but that, again, is below where we started. It will move everything that can be moved. RAM will be moved along, which may not hurt anything; then the IA chips will be moved, which may scramble colors or make the display go crazy; then it will try to move ROM, which won't work because ROM can't be changed; then it will wrap around to zero-page and move everything there, which is fatal to the system. Eventually, it will collapse before reaching SOV since it destroys its own working pointers.

All this could have been avoided if the programmer had loaded the machine language program first, and then loaded the BASIC program. The SOV would be placed behind the BASIC program, which is where it belongs in this case.

Quiet Interlude

It's easy to see how the problem occurs, once you understand about the SOV and its role. But if you don't understand the SOV, the results can

shake your self-confidence. Many programmers have given up on machine language because of a bad experience with SOV.

It works this way The student writes a perfect program into the cassette buffer and saves it using the machine language monitor Later, with a BASIC program in place, the student recalls the program and inadvertently moves SOV to an impossible location. When BASIC runs, the variables will start to be written behind the machine language program, ahead of the BASIC program. As more and more variables come into play, they creep relentlessly toward the BASIC coding.

Our eager student—with a perfect machine language program and a perfect BASIC program—now decides to say RUN. The BASIC program runs for a while, and then grinds to a halt, usually with a crazy screen or reporting an error in a nonexistent line. We know what's happened, of course: the variables have started to write over the BASIC program. But our unfortunate student doesn't know that. The command $LIST$ is entered, and out comes nonsense

What goes through the programmer's mind at this time? "I was so sure that the program is correct [in fact, it is]; but it's so bad that it's destroyed memory! I suppose that machine language is much more difficult than I thought."

And the student loses hope and gives up, not knowing that there's only one small piece of information needed to fix everything up. This is only one of the things that might go wrong when the SOV pointer is improperly placed; even an attempt to change or save a BASIC program can cause system failure.

Such experiences destroy confidence. They are responsible for the myth that machine language is hard and only super-clever programmers can cope with it.

The Machine Language Monitor $SAVE$

Now that we're becoming aware of the SOV pitfall, we're ready to discuss how to save a program in machine language. You probably understand why I've been delaying this command until this time. The MLM save command typically goes

```
.S "PROGRAM",01,033C,0361
```

This would be the tape format. The command is $.S$ and is followed by the program name. The device is tape, so we type 01—be sure to give two digits. Next comes the beginning address (in the example $033C$)

followed by the end address plus one. In the example, the last location saved will be $□3b□. For disk saves, we might want to add the drive number:

```
.S "□:PROGRAM",□8,□33C,□3b1
```

These programs, once saved, may be loaded directly from BASIC, but watch the SOV carefully. VIC-20 and Commodore 64 BASIC LOAD commands should contain the extra field to defeat relocation: LOAD "PRO-GRAM",8,1 will insist that the program load back into the same memory locations from which it was saved

More on LOAD

There is a machine language .L command to do a program load without changing any pointer (especially SOV). There are a number of different machine language monitors around, and the .L command does not work the same way on all of them. You might check out the one you are using: ideally, the .L command (format: .L "PROGRAM",□1) should bring back the program wthout relocation.

The .L command is of limited value. A program user often cannot be expected to load up a machine language monitor and use it to go through a .L load sequence. The program should take care of things for the user.

We have been careful to say that the BASIC LOAD command changes the SOV *when given as a direct command*. If a LOAD command is given from within a program, SOV is not changed; but there's a new item to be taken care of.

Programmed LOAD has been carefully designed to perform a function called "chaining." That's a BASIC technique, and not within the scope of this book. Chaining, however, has two important characteristics:

1 No pointers are affected. The program will not lose any variables when it performs a LOAD. That's good: we will not lose any of our computations.

2 Once a LOAD is complete, the BASIC program will resume execution *at the first statement* It will not continue from where it left off; it will go back to the beginning. For our application, that's bad; we seem to have lost our place in BASIC.

If we understand the problem that item 2 creates, we can easily fix it by using item 1. Here's an example to illustrate the problem: we have a program on disk written for the cassette buffer called "ML", and we want to have a BASIC program bring it in. We could code as a first line: 1□□ LOAD "ML",8—but we'd have a problem. First, the program would load

ML. Then it would go back to the beginning and load ML. Then it would go back to the beginning . and so on. This is not satisfactory. Let's use rule 1 to fix everything:

```
100 IF A=1 GOTO 130
110 A=1
120 LOAD "ML",8,1
130 ... continues
```

When we say RUN, the first line is executed. A is not equal to one, so we continue on line 110. A is set to one, and line 120 causes a load of the desired program. BASIC goes back to the beginning, but all variables are preserved, so A is still equal to 1. Line 100 tests A and goes to line 130, the next statement beyond the load. Everything works as required. If there are multiple LOADs, line 100 might be changed to 100 ON A GOTO 130,150,170 . . . as necessary.

Caution: we are discussing the programmed LOAD command only in the context of loading machine language modules. If you want to have a program load in another BASIC program (chaining or loading) the above rules still apply but may need to be used differently.

Other SOV Blunders

We have discussed the horrible results of loading a machine language program into the cassette buffer (using a direct command) *after* BASIC has been loaded. By now, we should have learned to avoid making this mistake. What about programs stored in other areas, such as high memory or after BASIC?

Suppose we want to place a program into high memory, either by moving the top-of-memory pointer down to make room, or by using the spare RAM at $C000 to $CFFF of the Commodore 64. We also have a BASIC program to load. Will loading in the wrong order harm SOV?

The answer is yes, although the problem is not so severe. You can see that after loading a program to high memory using a direct command, SOV will be positioned immediately above it. But that's too high—there's no room for variables and we'll get an OUT OF MEMORY error for almost anything we do.

Obviously, we can't leave SOV in the upper stratosphere. We must load the high memory first, and then the BASIC program. The second load will straighten out the SOV pointer. If you try this, you'll find that it is necessary to fix up the top-of-memory pointer and command NEW between the two

loads; you cannot even give the next LOAD command if you're apparently totally out of memory.

Review: Fixing Pointers

If in doubt, examine the pointers by displaying them with a . M command. For VIC/64/PLUS/4, the command would be . M 002B 003A; with PET/CBM, use . M 0028 0037; in either case, be sure that the start-of-variables pointer is set to a "sound" value.

As always, you can change an incorrect memory value—in this case, an incorrect vector—by moving the cursor back, typing over the values to be changed, and pressing RETURN.

After End-of-BASIC—Harmony

Suppose we place the machine language program behind the end-of-BASIC—that's the three zeros in memory—and move up the SOV so that variables won't disturb this program. How will everything work now?

Things will work very well indeed. This time, we need to load our BASIC program first; the SOV will go immediately behind BASIC. Then we may load our machine language program, and the SOV moves right behind it. The SOV is in exactly the right place, assuming we load in the right order. (If we don't, the variables will destroy our machine language program.)

Once our two programs are together, and we say SAVE, the combination program—BASIC and machine language together—will be saved. A little thought will reveal that memory from start-of-BASIC to just before start-of-variables contains everything we need. A subsequent load will bring everything back in, and position SOV to exactly the right place. We now have a "unit" program—BASIC and machine language working together, loading and saving as one program.

There's one small problem in this arrangement. Once we have married the BASIC and machine language programs, we must not change the BASIC program. If we added to or subtracted from this program, the machine language program would move up or down—the relocation of memory goes right up to SOV. The program might not be able to work in the new place, and, of course, our SYS commands would be wrong.

BASIC Variables

There are four types of entry in the BASIC variable table. All variables, regardless of type, occupy seven bytes; the first two bytes are the name,

and the remaining five bytes (not always fully used) contain the value or definition. The variable type is signaled as part of the name: high bits are set over one or both letters of the name to signal a specific type

SOV
SOA

EACH VARIABLE IS EXACTLY 7 BYTES LONG.
VARIABLES APPEAR IN THE ORDER IN
WHICH THEY ARE USED.

Figure 6.5

For example, if a floating point variable had a name AB, the name would be stored in the two bytes as $41, $42—the ASCII codes for A and B. The same would be true if the variable were named ABACUS, since only the first two letters of the name are kept. In contrast, if the variable were named AB%, meaning that it was an integer variable, the name would be stored as $C1, $C2. The ASCII codes are the same, but the high bit has been set over them. To complete the picture, a string variable named AB$ would be coded with the name $41, $C2—the high bit is set over the second character only

HIGH BIT SET FOR INTEGER VARIABLES AND FUNCTIONS

HIGH BIT SET FOR INTEGER AND STRING VARIABLES

NAME 2 BYTES	VALUE 5 BYTES
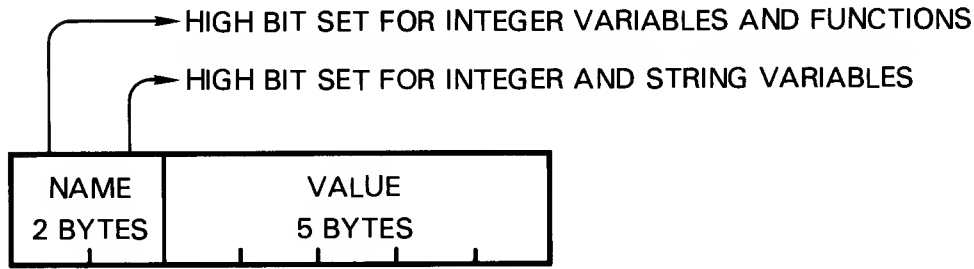
Figure 6.6

There's a fourth type of entry that can go into the variable table, but it's not a variable: it's a function definition. If we give the variable command DEF FNA (. . . an entry will be made in this table. It will be distinguished by the high bit being set over the first character only.

String variables use only three of the five bytes provided; the first byte signals the length of the string, and the next two bytes give the string's address. This group of three bytes is called a *descriptor*.

There are two types of numeric variables: floating point and integer. Floating point variables use all five bytes; integer variables use the first two bytes only. It's possible to extract the value from a floating point variable and put it to work, but it's not a simple procedure. A description of how to do this is given in Appendix F. In contrast, it's quite easy to take the value from an integer variable and use it.

Let's try an example. Type NEW, followed by A = 5 : B% = 5. This creates two different variables: A and B%. Now go to the machine language monitor. The variables should be near the start-of-BASIC, but if you wish you can find their exact address by examining the SOV pointer ($2A/$2B on PET/CBM, or $2D/$2E on VIC, Commodore 64 or PLUS/4). On the Commodore 64, we might find that the variables start at $0803; to display both of them, we type .M 0803 0810. We see the floating point variable, A:

    41 00 83 20 00 00 00

The first two bytes are the name—41 is ASCII for A, and the zero signifies no second letter—but where's the 5? Embedded within the 83 20 00 00 00, that's where; and it's a good deal of work to extract the 5 for further processing.

Behind this variable, we see the integer variable, B:

    C2 80 00 05 00 00 00

Hex C2 is the ASCII for the letter B (42) with the high bit set. 80 is zero with the high bit set—again, there's no second letter. The value is in the next two bytes, and it's easy to read. The last three bytes are not used.

Which is easier for machine language to interface with? Obviously, the integer variable. It's often quite suitable for the program work at hand: counting characters, setting pointers, and similar tasks.

Exchanging Data: BASIC and Machine Language

If BASIC and machine language wish to pass data back and forth, there are several approaches. Perhaps the simplest is to have BASIC POKE the values into a given location, and machine language load them as needed; in the opposite direction, machine language will store the values and BASIC will PEEK them.

Another method is more sophisticated. BASIC variables are stored in

memory: why can't a machine language program go after the variables exactly where they lie and extract their value or change them? It sounds like a good idea.

By now, we know how to ask machine language to search for a specific BASIC variable. Given the name, we can get the address of the first variable from the SOV pointer and store it as an indirect address. Using indirect, indexed addressing and stepping the Y register from 0 to 1 we can see if the name matches. If not, we add seven to the indirect address to take us to the next variable. If it does match, our indirect address is set up at the start of the variable; we can set Y to 2, 3, 4, 5, and 6 and extract the whole value. If the variable is type integer, we need only extract the first two bytes (Y = 2 and 3). If the variable is not in the variable table, we'll step our indirect address until it matches the start-of-arrays; at that point, we know that we have missed the variable

For a small number of variables, there's a short cut. Variables are placed into the variable table in the order in which they are defined: whichever variable is defined first in the BASIC program will be first in the variable table. So if we arrange for our variables to be defined in a certain order, we can streamline our machine language search to "first variable," "second variable," and so on, with no need to examine the names

Let's take this one step further. If we want to use the first variable, all we need to have is the address of the first variable somewhere in zero-page so that we may use it as an indirect address. *We already have that address*—it's the SOV, the start-of-variables, and it's there pointing helpfully at the first variable for us. By increasing the value of Y appropriately, we can reach beyond the first variable and into the second or, for that matter, the third or the thirty-sixth.

Project: We plan to place the machine language program behind the end-of-BASIC. *This will vary, depending on the machine being used*. The following code shows the correct addresses for the Commodore 64. Refer to Appendix E for other machines.

First, let's do our BASIC coding to estimate its size. We need to guess at the location of the end-of-BASIC so as to place our machine language program. This program will ask machine language to take a value, V%, and multiply it by ten. Remember to say NEW. We write the BASIC program as follows:

```
100 V% = 0
110 FOR J = 1 TO 5
120 INPUT "VALUE";V%
```

```
130 SYS ++++
140 PRINT  "TIMES TEN" =";V%
150 NEXT  J
```

It seems likely that our BASIC program will occupy less than 127 bytes. We may check this later, but it seems safe to plan to start our machine language program at around 2049+127,or 2176 (hexadecimal 880). On that basis, we may change line 130 to SYS 2176. *Do not* try to run the program yet.

At this point, we could save the BASIC program to tape or disk and develop the machine language program. This would allow us to refine each of the two parts independently. For the sake of brevity—and because our example is an easy one and won't need touching up—we'll write the machine code directly into memory.

Switch into the machine language monitor. Assemble the following code:

```
.A  0880  LDY  #$02
.A  0882  LDA  ($2D),Y
.A  0884  STA  $033C
.A  0887  STA  $033E
.A  088A  LDY  #$03
.A  088C  LDA  ($2D),Y
.A  088E  STA  $033D
.A  0891  STA  $033F
```

We have now extracted two bytes from the first variable, V%. The high byte has been stored at both $033C and $033E; we'll see why in a moment. The low byte of the value has gone to $033D and $033F.

Project for enthusiasts: You might be able to code the above more compactly by more effective use of indexing.

```
.A  0894  ASL  $033D
.A  0897  ROL  $033C
.A  089A  ASL  $033D
.A  089D  ROL  $033C
```

We have multiplied the contents of $033D/$033C by two, and then we have multiplied it by two again. These locations now contain the original value times four. Note that we ASL the low byte and then ROL the high byte. Perhaps we should be checking for overflow; but let's trust the number to be within range for now.

Since we have the original number times four in $033D/$033C, we can

add it to the original number in $033F/$033E to get the original number times five:

```
.A   08A0   CLC
.A   08A1   LDA   $033D
.A   08A4   ADC   $033F
.A   08A7   STA   $033D
.A   08AA   LDA   $033C
.A   08AD   ADC   $033E
.A   08B0   STA   $033C
```

Now locations $033C/$033D contain the original number times five. If we double the number one last time, we'll have the value times ten:

```
.A   08B3   ASL   $033D
.A   08B6   ROL   $033C
```

We have multiplied the number by ten. Now let's put it back into the variable

```
.A   08B9   LDY   #$02
.A   08BB   LDA   $033C
.A   08BE   STA   ($2D),Y
.A   08C0   LDY   #$03
.A   08C2   LDA   $033D
.A   08C5   STA   ($2D),Y
.A   08C7   RTS
```

The numbers go back exactly the same way we drew them out. We must be careful to keep the high and low bytes correct. Integer variables have the high-order byte first, followed by the low-order byte; this is exactly the reverse of the way we use 650x addresses.

We must perform one more task before wrapping up the program. We must change the state-of-variables pointer to a location above the machine language program. That would be $08C8, and so we display the SOV pointer with .M 002D 002E and change the pointer to

.:002D C8 08 .. ..

Check ... disassemble ... and then back to BASIC. List, and you'll see your BASIC program again. There's no sign of the machine language program, of course, but SAVE will now save everything together.

RUN the BASIC program. Enter numbers as requested. Confirm that they are multiplied by ten.

You may recall that our machine language program does not check for

overflow. RUN the program again, and see if you can find the highest number that can be multiplied by ten without error. What happens at time of overflow? Is it what you expected?

Project for enthusiasts: Can you add checks for overflow to the above program? You must decide what to do if overflow occurs: print a message; set the value to zero; or whatever you decide. But you shouldn't stop the program or break to the monitor. Such a thing would upset the program user. Your program will be longer. Don't forget, therefore, to change the SOV pointer at $2D/$2E so that your program is safe from variables

Things You Have Learned

—Small machine language programs can be conveniently written and checked out in the cassette buffer. We have been doing this during the exercises This area is not satisfactory for large programs, or programs we want to save on tape.

—Programs can take up semi-permanent residence near the top-of-BASIC memory: the top-of-memory pointer needs to be moved down to protect it These programs often need a separate "setup" to place them.

—Programs can be placed behind the end-of-BASIC, which is marked by three consecutive zero bytes in memory. The start-of-variables pointer must be increased so that variables don't write over the program Care must be taken not to change the BASIC program after this is done.

—The VIC-20 frequently has the start-of-BASIC moved up to make room for video information in lower memory. As long as we're moving this pointer, we might move it a little further and make room for some machine code.

—The Commodore 64 has an unused block of RAM at addresses $C000 to $CFFF; check to see that no other programs are using this area.

—The start-of-variables pointer is intimately tied in with BASIC's $SAVE$ and $LOAD$ commands. It is extremely important to ensure that any $LOAD$ sequence leaves this pointer in a safe place, so that variables cannot write over program code and thus cause program destruction.

—Machine language monitor .S (save) and .L (load) commands can be used for staging programs in various parts of memory. Again, great care should be taken to ensure that the pointers are sound after the use of such instructions.

—A BASIC program may contain $LOAD$ commands that will bring in any of the following: a different BASIC program, a machine language program, or data. Again, careful handling is needed.

—BASIC variables are of three major types: integer, real (floating point), and string. Machine language programs are capable of reading and using any of them; in particular, integer variables are quite straightforward.

—If we want, we can simplify the task of searching for BASIC variables by deliberately creating them in a certain sequence.

Questions and Projects

Write a simple BASIC and machine language program set that allows BASIC to input a number less than 256; POKE it somewhere in memory; call machine language that will divide the number by two; PEEK it back and print it.

A program that brings in other programs is called a "boot," or, more accurately, a *bootstrap* program. Write a simple BASIC boot program to bring in a previous program exercise that was located in a cassette buffer (say, the program from Chapter 2 that printed HELLO), and then call it with a SYS.

Bootstrap programs are especially popular with VIC, Commodore 64, and PLUS/4 for bringing in chunks of data such as sprites, new character sets, or whole display screens of information. You might like to try your hand at setting up such a system.

Try your hand at this. I have a BASIC program that reads

```
100 X = 5
110 SYS ...
120 PRINT A
```

Write the machine language to be called by the SYS so that it changes the name of the variable X to A. Caution: this may be fun, but it's dangerous in real programs since you may end up with two variables that have the same name.
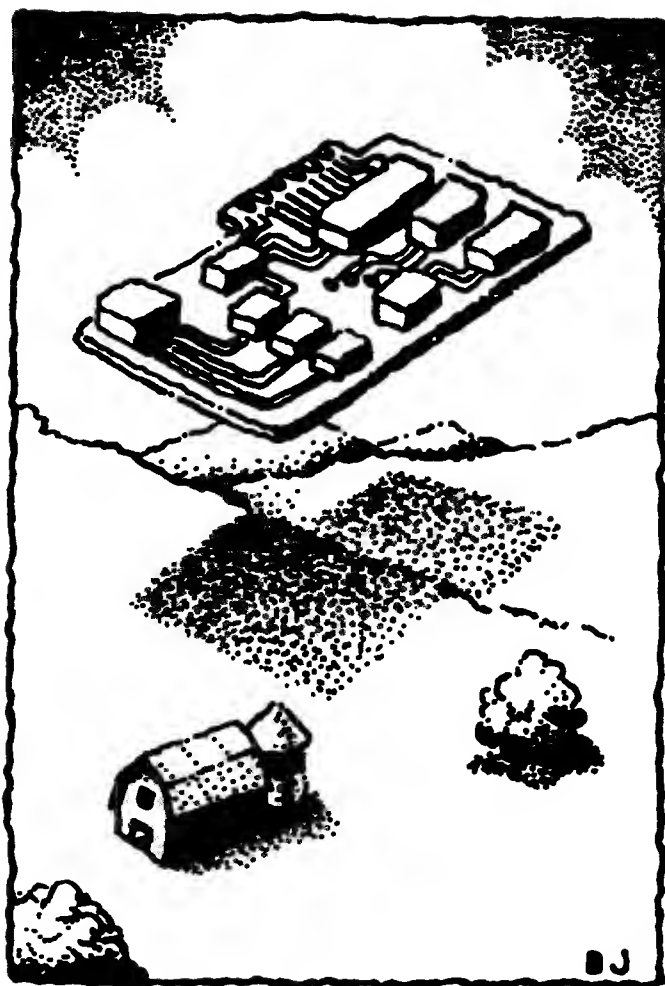
7

Stack, USR, Interrupt, and Wedge

This chapter discusses:

- The stack for temporary storage
- USR: an alternative to SYS
- Interrupts: IRQ, NMI, and BRK
- The IA chips· PIA and VIA
- Infiltrating BASIC: the wedge

A Brief Intermission

If you have been following along and performing the various projects, you should know a great deal about the principles of machine language. You should be capable of trying your hand at a number of small projects, and investigating areas that may be of special interest.

This is a good time to stop and take stock. The remaining chapters are "icing on the cake" . . . they give extra detail and fine tuning on aspects of machine language. If you feel uncertain about any material covered so far, go back Fix the fundamentals firmly in focus before you proceed and plunge into ponderous points of interest.

Temporary Storage: The Stack

The stack is a convenient place to put temporary information. It works like a stack of documents: you may drop (or "push") an item onto the stack; when you take an item back again (or "pull"), you'll get the last one that you put there. Formally, it's called a last-in, first-out (LIFO) discipline; it's natural and easy to understand.

The important rule to keep in mind about the stack is: "Leave these premises as clean as when you found them." In other words, if you push three items onto the stack, be sure you pull those three items back off again. Don't ever branch away and leave the stack littered.

The stack is in memory at page 1. The stack pointer (SP) is one of the items displayed in the register. To look for the information on the stack, you must add $\$0100$ to the value to get the next available stack position. As an example, if the SP shows a value of $\$F8$, the next item to go on the stack will go into address $\$01F8$; the moment we put an item onto the stack, the pointer will move *down* so that it becomes $\$F7$.

As the stack is filled, the stack pointer goes *down*. As the items are brought back out of the stack, the stack pointer goes *up*. A low value in the stack pointer means a full stack. a value below $\$40$ signals trouble.

The 650x chip itself doesn't give the stack any special treatment. If a machine language program—probably because of a coding error—wanted to push one thousand items onto the stack, that would be OK as far as the microprocessor was concerned. The stack would never leave page 1: as the stack pointer went down beyond zero, it would wrap around to $\$FF$ and keep going. You'd never get those thousand distinct items back, of course. Similarly, if a program wanted to pull a thousand items from the

SP

| F8 |

USED	01FF
USED	01FE
USED	01FD
USED	01FC
USED	01FB
USED	01FA
USED	01F9
FREE	01F8

NEXT ITEM
PUSHED WILL GO
TO ADDRESS $01F8

NEXT ITEM
PULLED WILL COME
FROM ADDRESS $01F9
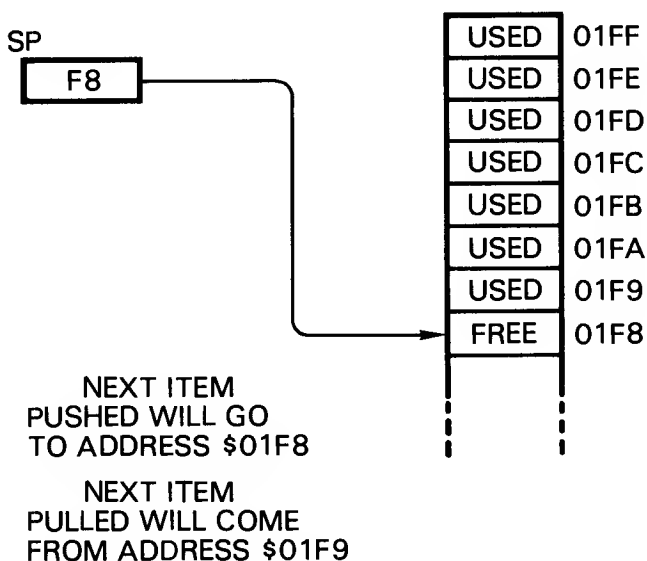
Figure 7.1

stack—whether or not they had been put there before—the processor would happily move the stack pointer round and round page 1, delivering bytes. There would only be 256 different values delivered, of course, but the processor doesn't care.

Within the BASIC environment, the stack pointer starts around $FA (the first item will go into the stack at address $01FA), and goes down from there. When the stack pointer goes below about $40, BASIC will signal OUT OF MEMORY. That's over 160 available locations on the stack, plenty of room for most applications

PHA *(push* A) *and* PLA *(pull* A)

How may we use the stack? Suppose we have a value in the A register and in a moment we will want to use it. First we need to print something, and the character to be printed must be loaded into the A register How can we put away the value in A and bring it back later? We could slip it into another register with a transfer instruction (TAX or TAY) and bring it back from there; or, we could store it into memory and load it back. Alternatively, we could PUSH the A register (PHA) to the stack and PULL (PLA) the value back later.

Again, let's do an example. Suppose the A register contains 5, and the

stack pointer is at $F3. If the program says PHA, the value 5 is stored at address $01F3, and the stack pointer changes to $F2. Later in the program, we encounter the instruction PLA: the stack pointer moves back to $F3 and the value 5 is read from address $01F3 and placed into the A register.

It's a handy way to put away a value in A for a moment.

PHP (push processor status) and PLP

Sometimes when we are writing a program, we want to test for a condition now but act on the result of that test later. We can arrange to do this by putting the flags away for the time being, and then bringing them back when we want to test the flags. We use the instruction PHP (push the processor status word) to place all the flags on the stack, and PLP (pull the processor status word) to restore the flags to the status register (SR).

Why would we need to do this? Perhaps an example will illustrate. Suppose we are reading a file of customer purchases, and as we input a data item, we discover that this is the last one—it's the end of the file. That means that we want to close the file and summarize the customer's activity—though not just yet. First, we must handle the item of information that we have input. So we can "stack" our end-of-file information, handle the last record in the same way as previous records, then bring back the status to see whether it's time to close the file and print the totals. We'll be using PHP and PLP for exactly this kind of task in the next chapter

PHA and PHP both put exactly one item onto the stack; PLA and PLP pull one item. There are other commands that handle more than one stack location.

JSR and RTS

We know these commands. What are they doing here?

When a JSR command is executed, the *return address* is placed onto the stack. When an RTS command is executed, the return address is picked from the stack, and that's where the program returns to.

More precisely, when a JSR occurs, the processor places onto the stack the *return address minus one* as two bytes; the high-order part of the address goes to the stack first. When an RTS is encountered, the processor takes the two bytes from the stack, *adds one*, and then proceeds from the address so formed

Example: If address $0352 contains the command JSR $033C, the

following events occur. The return address would be $0355, the instruction directly behind the JSR; but an address of $0354 is calculated—the 03 goes to the stack first, and the 54 below it. The subroutine at $033C now starts to run. Eventually, it encounters an RTS. The values 54 and 03 are pulled from the stack and formed into address $0354; one is added, and the processor resumes execution at address $0355.

You hardly need to know this. We have been using subroutines for some time without knowing that all this happened. But sometimes, it's useful to be able to examine the stack, asking, "Who called this subroutine?" The answer is there.

Interrupts and RTI

There are three types of interrupt: IRQ, NMI, and the BRK instruction. IRQ (interrupt request) and NMI (non-maskable interrupt) are pins on the 650x. A suitable signal applied to the appropriate pin will cause the processor to stop what it's doing and run an *interrupt routine*. The BRK instruction might be thought of as a fake interrupt—it behaves in a similar manner to IRQ.

When an interrupt signal occurs, the processor completes the instruction it is currently working on. Then it takes the PC (the program counter, which contains the address of the next instruction) and pushes it onto the stack, high byte first. Finally, it pushes the status register to the stack. That's a total of three bytes that go to the stack.

The processor then takes its execution address from one of the following locations:

IRQ or BRK—from $FFFE and $FFFF

NMI   —from $FFFA and $FFFB

Whatever value is found in these pointers becomes the interrupt execution address· the processor starts to run at this address. Eventually, the processor encounters an RTI instruction. The status register and the PC address are taken from the stack, and the interrupted program resumes where it left off.

Note that the address on the stack is the return address. This differs from JSR/RTS, where the return address minus one is stored.

On all Commodore machines, the IRQ strikes about sixty times a second. The NMI is unused (but available) on PET/CBM; it isn't available in the

264 series; and on VIC-20 and Commodore 64, it is used for the RE-STORE key and for RS-232 communications.

The BRK command can be distinguished from the IRQ signal by means of a bit in the status register. Bit 4 is the B, or break flag: if it's set. the last interrupt was caused by a BRK and not by an IRQ.

Later, we will discuss using the interrupt routines for our own programming. By the time we can "catch" the interrupt, several more things will have been pushed to the stack: the A, X, and Y registers. This is done by a ROM program, not the processor; but it will prove handy since we can use these registers, safe in the knowledge that they will be restored at the end of the interrupt.

Mixing and Matching

The processor uses the stack mechanically. If we know how to manipulate the stack, we can use it for surprising things. For example, an RTS can be given even though there was no subroutine call; all we have to do is prepare the stack with the proper address. Try to figure out what the following code will do:

```
LDA #$24
PHA
LDA #$68
PHA
RTS
```

This coding is identical to JMP $2469. We have placed a "false return address" onto the stack, and RTS has removed it and used it. This may not seem very useful, since we could easily have coded the JMP $2469 directly. But look at the following code:

```
LDA TABLE1, X
PHA
LDA TABLE2, X
PHA
RTS
```

The principle of coding is the same, but now we can "fan out" to any of several different addresses, depending on the value contained in X.

USR: A Brother to SYS

We have used SYS a number of times. It means, "Go to the address supplied and execute machine code there as a subroutine." USR is similar

in many respects: it means, "Go to a fixed address and execute machine code there as a subroutine." The fixed address may be POKEd into the USR vector. On most Commodore machines this is at addresses 1 and 2; on the Commodore 64, it's at addresses 785 and 786 (hex 0311 and 0312).

There's another difference that seems important at first. SYS is a command; USR is a function. You cannot type the command USR(0)—all you'll get is SYNTAX ERROR. You must say something like PRINT USR(0) or X = USR(0), where USR is used as a function. It seems as if SYS was meant to connect to action programs, and USR was meant to link to evaluation programs. In reality, the difference in usage is not that great.

Whatever value is within the parentheses—the argument of the USR function—is computed and placed into the *floating accumulator* before the USR function is called. The floating accumulator is located at $5E to $63 in most PET/CBM computers, and at $61 to $66 in VIC-20, Commodore 64, and PLUS/4. Floating-point representation is complex, as we have hinted in Chapter 6. Most beginning programmers prefer to leave this area alone and pass values through memory POKEs or integer variables.

When the USR function returns control to BASIC, the function value will be whatever is in the floating accumulator. If we have not modified it, this will be the same as the argument, so that in many cases PRINT USR(5) would print a value of 5.

Interrupts: NMI, IRQ, and BRK

We have mentioned the mechanical aspects of interrupt Now let's look at how to use the interrupt for simple jobs.

The IRQ connects through a vector in RAM; if we change the address within the vector, we will change the address to which the interrupt goes. The interrupt vector is located as follows.

Most PET/CBM: 0090-0091 (decimal 144-145)

VIC/Commodore 64: 0314-0315 (decimal 788-789)

Before we change this vector, we should realize something quite important: the interrupt does a lot of work sixty times a second. It updates the clock, checks the RUN/STOP key, gives service to the cassette motors, flashes the cursor, and handles keyboard input. If you thoughtlessly change the IRQ vector, it will stop doing these things; and it's hard to handle a

computer when it has a dead keyboard. You could try to program all these functions yourself; but there's an easier way.

Suppose we use the vector to temporarily divert to our own program, and at the end of our program we allow the interrupt to continue with whatever it was going to do anyway. That way, our program would get service sixty times a second, and the usually interrupted jobs would still get done.

It's not hard to do, and we can achieve many interesting effects by diverting the interrupt. Remember that the interrupt runs all the time, even when no BASIC program is running By playing with the interrupt, we can make a permanent computer system change that is in effect even when no programs are in place

Care must be taken in changing an interrupt vector. Suppose we are beginning to change the two-byte address; we have changed the first byte, and suddenly, the interrupt strikes. It will use an address that's neither fish nor fowl: half is the old address, and half is the new In such a case, it's likely that the interrupt will become confused; and if the interrupt is confused, the whole computer is in trouble. We must find a way to prevent interrupt from striking when we change the vector

We could do this in machine language: before a routine to change the IRQ vector, we could give the instruction SEI (set interrupt disable). After this instruction is given, the IRQ cannot interrupt us. We may set the vector and then re-enable the interrupt with the instruction CLI (clear interrupt disable) Be sure that you do this, since the interrupt routine performs many vital functions. We may say that we have *masked off* the interrupt in the time period between execution of SEI and CLI. The NMI interrupt, however, is *non-maskable*, and SEI will have no effect on it.

There's a second way of turning off the interrupt—that is, by shutting off the *interrupt source* Something makes an interrupt happen—it might be a timer, it might be an external signal, or it might even be a screen event. Whatever it is, we can get to the source of the interrupt and disconnect it.

Almost all interrupt signals are delivered through an IA (interface adaptor) chip; and these chips invariably allow the path of the interrupt signal to be blocked temporarily. We'll discuss the IA chips later, for the moment, the normal interrupt signals can be blocked with the following actions:

Commodore 64 Store $7F into address $DC0D (POKE 56333,127) to disable; store $81 into the same address (POKE 56333,129) to re-enable.

VIC-20: Store $7F into address $912E (POKE 37166,127) to disable; store $C0 into the same address (POKE 37166,192) to re-enable.

PET/CBM: Store $3C into address $E813 (POKE 59411,60) to disable; store $3D into the same address (POKE 59411,61) to re-enable.

It goes without saying that the above POKEs should not normally be given as direct commands; the first POKE in each case will disable the keyboard (among other things), and you won't be able to type the restoring POKE.

A warning about interrupt programs: changing the IRQ vector is likely to make it difficult to load and save programs. You may need to put the vector back to its original state before you attempt any of these activities.

An Interrupt Project

The following project is written for the Commodore 64 only. The equivalent coding for PET/CBM may be found in Appendix E.

Let's write the coding for the interrupt itself. Sixty times a second, we'd like to copy the contents of address $91 to the top of the screen. Here goes:

```
.A  033C  LDA  $91
.A  033E  STA  $0400
.A  0341  JMP  ($03A0)
```

Why the indirect jump? We want to "pick up" the regular interrupt routine, but we don't know where it is yet. When we find the address, we'll put it into locations $03A0/$03A1 so that the indirect jump will link things up for us.

Now let's write the routine to enable the above interrupt coding. First, let's copy the interrupt address from $0314 into the indirect address at $03A0:

```
.A  0344  LDA  $0314
.A  0347  STA  $03A0
.A  034A  LDA  $0315
.A  034D  STA  $03A1
```

Now we are ready to put the address of our own interrupt routine (at $033C) into the IRQ vector:

```
.A  0350  SEI
.A  0351  LDA  #$3C
.A  0353  STA  $0314
```

```
.A  0356  LDA  #$03
.A  0358  STA  $0315
.A  035B  CLI
.A  035C  RTS
```

We will enable the new interrupt procedure by a SYS to $0344, above (SYS 836). Before we give that command, let's write the coding to put everything back·

```
.A  035D  SEI
.A  035E  LDA  $03A0
.A  0361  STA  $0314
.A  0364  LDA  $03A1
.A  0367  STA  $0315
.A  036A  CLI
.A  036B  RTS
```

As you can see, we put the original address back, copying it from the indirect address area where it was saved.

Once this code is in place, disassembled, and checked, you may return to BASIC. SYS 836 will invoke the new interrupt code; SYS 861 will turn it off. Note that the character (a copy of the contents of address $91) appears at the top left of the screen. The character seems to be affected by pressing some keys; can you establish how many keys are involved?

Some models of Commodore 64 may print blue-on-blue when screen memory is POKEd, as we are doing now. If so, the character may not always appear in the left-hand corner. **Project for enthusiasts**: Fix this problem by storing a value into the color nybble table at address $D800.

The I A Chips: P I A, V I A, and C I A

The interface adaptor (IA) chips are richly detailed. To understand them fully, you'll need to read the specifications in some detail. Here, we'll give their main functions.

PIA stands for peripheral interface adaptor, VIA for versatile interface adaptor, and CIA for complex interface adaptor. There is speculation among Commodore owners that the next interface chip will be called "FBI."

The functions performed by an interface adaptor are:

1   Event latching and interrupt control We have noted that these chips can be manipulated to block the interrupt signal In fact, they do more than "gating" the signal—allowing it through to the processor's IRQ trigger or alternatively

blocking it. They also often *latch* a signal into an *event flag*, sometimes called an *interrupt flag*.

Latching is important A triggering event may be brief, so short, in fact, that the original signal causing interrupt might go away before the processor can look at it An I A event flag locks in the signal and holds it until the program turns it off.

NEAR "THE IA CHIPS . . . ''
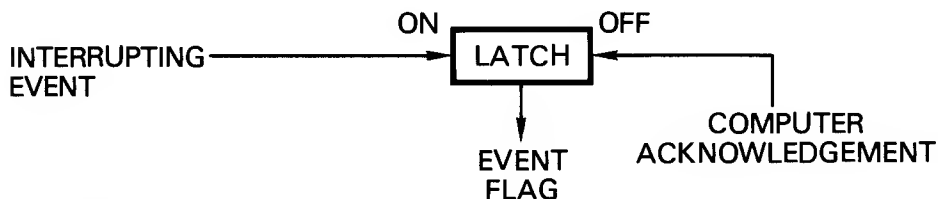
Figure 7.2

If an event has time importance—that is, if the event's timing must be accurately measured, or if the event flag must be cleared quickly so as to allow for the detection of a new event—we may link the event flag to the interrupt line If we do so, the occurrence of the event will cause the processor to be interrupted. We must write coding linked to the interrupt routines to detect this event, clear the flag, and do whatever processing is needed We set up this link to the interrupt line by means of a register usually called the *interrupt enable register*.

On the other hand, the event might not be particularly time critical. In this case, you can simply check the appropriate event flag from time to time. When the event occurs, you may then clear the flag and handle it. No interrupt is needed. Even when an event flag is not connected to the interrupt, it may be called an interrupt flag; don't let the terminology confuse you.

Whether or not you handle these events through interrupt sequences, it's important to know that it's your job to turn the event flag off. The flag will hold the signal until it's turned off—and it usually won't turn off unless your program takes some action to do this.

The various flags are triggered by timers or external signals You can read a flag's state by checking the *interrupt flag register*. Several flags will be packed together in this register; as always, you will use the logical operators— AND, ORA, or EOR—to extract or modify the particular flags in which you are interested. You may also use the IFR (interrupt flag register) to clear the flags

2. Timing. Certain addresses within the I A chip are often assigned as "timers." These timers count down; in other words, if we place a value of $97 into a

timer and look at the value immediately, we might find that it has gone down to $93 Timers come in many shapes and sizes—again. check the chip reference for details—but most of them toggle an interrupt flag when they have counted down to zero. As discussed, you may choose whether or not this flag will really cause an interrupt signal.

3. Input/output Certain addresses within the IA chip are connected to "ports," which extend outside the computer Thus, the computer can detect external events or control external devices Output signals are usually latching in nature: in other words, a store command might be taken to mean, "turn on port 5 and leave it on."

Tips on IA Chips

Many addresses within an IA chip have a different meaning, depending on whether they are being written to (stored) or read (loaded). Watch for this when you are reading the chip specifications.

Often, the action required to turn an interrupt flag off is odd. It looks like the kind of thing you should do to turn the flag on. Keep in mind that a flag may be turned on only by the external activity to which it is linked. So, although it may seem odd to turn the flag in bit zero off by storing a value of 1 (which would seem to want to turn bit zero on), don't worry. You'll get used to it.

The IER (interrupt enable register) is often a source of problems. In many cases, the high bit of a value we are storing has a special meaning: if it's set, the other bits will cause the appropriate interrupt connections to turn on; if it's clear, the other bits will cause the appropriate interrupt connections to be turned off. You may recall that we shut off the Commodore 64 interrupt by storing $7F into address $DC0D. This may seem odd: we're storing a binary value of $01111111, which might seem to be turning bits on. In fact, the high bit of zero signals that all the remaining bits are"turn off" signals: so the value causes all interrupts to be blocked.

Infiltrating BASIC: The Wedge

In zero-page, there's a subroutine that the BASIC interpreter uses frequently to obtain information from your BASIC program. It's used to get a character from your BASIC program, and to check it for type (numeric, end-of-command, or other).

The routine is normally entered at either of two points: CHRGET, to get the next character from your BASIC program; and CHRGOT, to recheck the last character. The subroutine is located at $0070 to $0087 in most

PET/CBM computers, and at $0073 to $008A in VIC-20 or Commodore 64. You may disassemble it there if you wish. The coding is described below.

Since CHRGET is in different locations, depending on the machine, the following coding is shown with *symbolic addresses* That is, instead of showing the hex address value, the address is given a name, or symbol. Thus, CHRGOT might represent address $0079, CHRGOT+1 would represent address $007A, and so on.

```
CHRGET INC CHRGOT+1
       BNE CHRGOT
       INC CHRGOT+2
CHRGOT LDA xxxx
```

This subroutine is *self-modifying*. that is, it changes part of itself as it runs. That's not always a good programming technique, but it works well here.

The first part of the subroutine adds one to the address used by instruction CHRGOT. This is a standard way of coding an address increment: add one to the low byte of the address; if that makes it zero, the low byte must have gone from $FF to $00, in which case, add one to the high byte.

The address loaded by CHRGOT is within your BASIC program, or within the input buffer if you have just typed a direct command. Before we follow the next piece of code. let's look at our objectives:

1. If we find a space, go back and get the next character
2. If we find a zero (BASIC end of line) or a colon (hex $3A, BASIC end-of-statement), we wish to set the Z flag and exit.
3. If we find a numeric, we wish the C flag to be clear; if we do not find a numeric, we wish the C flag to be set

```
CHRGOT LDA xxxx
       CMP #$3A
       BCS EXIT
```

If the character is a colon ($3A), we'll leave the subroutine with the Z flag set. That's one of our objectives. Here's part of another one: if the character is $3A or higher, it can't possibly be an ASCII numeric—numerics are in the range of $30 to $39.

```
CMP #$20
BEQ CHRGET
```

If the character is a space, we go back and get another character.

The following coding looks rather strange, but it's correct. After the two subtractions, the A register will be back where it started:

```
SEC
SBC #$30
SEC
SBC #$D0
```

After this, the A register is not changed; but the C flag will be set if the number is less than $30, which means that it is not an ASCII numeric. Additionally, the Z flag will bet set if A contains a binary zero. We have met all our objectives and may now return:

```
EXIT RTS
```

Breaking Into BASIC

Since BASIC comes to this subroutine often, we can infiltrate BASIC by changing this subroutine. Extra coding in this area is often called a "wedge" program. We must be very careful:

- We must leave A, X, and Y unchanged, either we must not use them or we must save them away and bring them back.
- We must not interfere with the flags.
- We must be careful not to slow BASIC down too much.

This is a tall order. The last requirement is often helped by two techniques: use the wedge to implement extra commands in direct mode only; and make use of a special character to identify our special commands.

In PET/CBM, we may choose to modify this subroutine in either of two places: near the beginning, in CHRGET; or after the LDA, in CHRGOT. Each location has its advantages. In the CHRGET area, we don't need to preserve the A register or status flags, since CHRGOT will fix them up for us. In the area following CHRGOT, we have the character we wish to examine in the A register.

But in either case, it's an exacting job.

VIC-20 and Commodore 64 have made the job much more easy by providing a vector at address $0308/$0309 that will give us control of the computer, if we wish, immediately before each BASIC command is executed. We still need to use due care, but we have much more latitude.

The address of the instruction at CHRGOT is often referred to as TXTPTR, the text pointer. This address always points to the BASIC command being

executed at the moment. If we want to participate in reading BASIC, we must learn to use $TXTPTR$ to get the information—usually by means of indirect, indexed addressing—and to leave this address pointing at a suitable place when we return control back to the normal BASIC handling programs.

Project: Adding a Command

Let's add a simple command to the VIC and Commodore 64 by using the $0308 vector. The ampersand (&) character isn't used in most BASIC programs, so we'll make it mean this: whenever you see the code "&", print ten asterisk (*) characters to the computer screen, followed by a carriage return.

As with our interrupt program, we'll copy the old address from $0308/0309 into an indirect address location, so that we can link up with the normal computer routines as necessary.

An important point· the vector will give us control, if we want it, with $TXTPTR$ positioned immediately before the next instruction. When we return control to BASIC, we must be sure that $TXTPTR$ is similarly positioned.

Here's our instruction "intercept":

```
.A 033C LDY #$01
```

We're going to use indirect, indexed addressing to "look ahead" at the instruction. Let's look, using $TXTPTR$ as an indirect address:

```
.A 033E LDA ($7A),Y
```

Since Y equals one, we'll look just beyond the address to which $TXTPTR$ is pointing:

```
.A 0340 CMP #$26
.A 0342 BEQ $0347
.A 0344 JMP ($03A0)
```

If the character is an ampersand, we'll branch ahead to $0347. If not, we'll connect through the indirect vector to the regular BASIC interpreter code:

```
.A 0347 JSR $0073
```

We may call $CHRGET$ to move the pointer along. Now $TXTPTR$ points squarely at the ampersand character. We are ready to print ten asterisks:

```
.A   034A   LDY   #$00
.A   034C   LDA   #$2A
.A   034E   JSR   $FFD2
.A   0351   INY
.A   0352   CPY   #$0A
.A   0354   BCC   $034E
.A   0356   LDA   #$0D
.A   0358   JSR   $FFD2
.A   035B   JMP   $0344
```

The above code prints an asterisk ($2A) ten times and then prints a RETURN ($0D). It then goes to the regular BASIC interpreter, which will look behind the ampersand character for a new BASIC command.

Now we need to set up the link to our program. We'll write the code to do this starting at $035E, so that SYS 862 will put the new command (ampersand) into effect:

```
.A   035E   LDA   $0308
.A   0361   STA   $03A0
.A   0364   LDA   $0309
.A   0367   STA   $03A1
.A   036A   LDA   #$3C
.A   036C   STA   $0308
.A   036F   LDA   #$03
.A   0371   STA   $0309
.A   0374   RTS
```

When you have completed and checked the code (remember this is for VIC and Commodore 64 only), return to BASIC. Type NEW and write the following program:

```
100 PRINT 3 4:&:PRINT 5+6
110 &
120 PRINT "THAT'S ALL"
```

If you type RUN, you will get a SYNTAX ERROR in line 100. We have not yet implemented our "ampersand" command. Type the command SYS 862. Now type RUN again. The ampersand command obediently prints ten asterisks each time it is invoked.

Infiltrating BASIC isn't an easy job. But it can be done.

Things You Have Learned

—The *stack* is located in page 1, from $01FF moving down to $0100. It is used for holding temporary information. A program may *push* information to

the stack, and then *pull* it back later The last item that has been pushed onto the stack will be the first item to be pulled back off

—Great care must be taken to ensure that your program pulls exactly the same number of items back from the stack as it pushed In particular, be sure that a branch or jump does not inadvertently omit a needed stack activity. A badly handled stack is often fatal to the program run

—PHA pushes the contents of A to the stack; PLA pulls from the stack into the A register. These two commands are often used to temporarily save A. PHP pushes the status register (SR); PLA pulls it back. These two commands are often used for "deferred decisions."

—JSR pushes a return address (minus 1) to the stack; RTS recalls this address. We may use JSR and RTS without needing to know the role the stack plays, since the two commands take care of the details for us.

—Interrupts, including the BRK instruction, push three items to the stack; RTI brings them back so that the interrupted program may resume

—USR is a function, as opposed to SYS, which is a command. USR goes to a preset address, takes a numeric argument, and can return a value. In practice, USR and SYS are used in quite similar ways.

—Commodore ROM systems contain coding for the interrupt sequences that cause the data registers—A, X, and Y—to be pushed to the stack, and a branch to be taken through an indirect address that the user can modify. Since interrupt is active virtually all the time, it may be used to create activities that are active even when no BASIC program is running.

—The various IA chips—PIA, VIA, and CIA—perform many different functions, including: recording events in latching flags and controlling interrupts, timing; and connecting input/output ports. The detailed specification sheets must be studied for these rather complex details.

—A subroutine called CHRGET is used frequently by the BASIC interpreter when a BASIC program is running. We may modify or add to this subroutine in order to add to or modify the BASIC language itself

Questions and Projects

If you redirect the interrupt vector to your own machine language program, you can copy all of zero page to the screen. Use indexing; start X at zero; and walk through the whole of zero page, loading the memory contents and storing (indexed again, of course) to the screen. Don't forget to connect up your code to the regular interrupt entry address.

You'll get a fascinating screen. There will be timers going, and as you type on the keyboard you'll see various inner values changing around. Enjoy the view.

It's sometimes suggested that a good way to pass information to a sub-

routine is to push the information onto the stack and call the subroutine. The subroutine can pull the information from the stack. What's wrong with this suggestion?

The above suggestion can be implemented, but it takes a lot of careful stack work. You might like to work through the logic needed to do this.

There are some utility programs which, when placed in the computer, allow a listing to be "scrolled." In other words, if the screen shows BASIC lines 250 to 460, the user can take the cursor to the bottom of the screen and continue to press the cursor-down key. New BASIC lines (following 460) will then appear. This is not an easy thing to code, but here's the question: do you think that this feature is done with a SYS command, a wedge, or an interrupt technique? Why?

A SYS command from BASIC is like a subroutine call; so it must place an address on the stack to allow RTS to return to BASIC. Take a look at the stack and see if you can determine what address is used to return to BASIC on your machine.

Intentionally Blank

8

Timing, Input/Output, and Conclusion

This chapter discusses:

- How to estimate the speed of your program
- Input and output from tape, disk, and printer
- Review of instructions
- Debugging
- Symbolic assemblers
- Where to go from here

Timing

For many applications, machine language programs seem to run instantaneously. The speed of the 650x is much greater than that of other devices, including the human user. The machine language program usually ends up waiting for something: waiting for the keyboard, waiting for the printer, waiting for the disk, or waiting for the human to read and react to information presented on the screen.

Occasionally, it may be important to get fairly precise timing for a machine language program. If so, the following rules of thumb may be kept in mind:

—All timing estimates are crude if the interrupt routines are still active. The effect of interrupt on timing can be crudely estimated by adding 10 percent to the running time.

—Remember to allow for loops. If an instruction within a loop is repeated ten times, its timing will need to be counted ten times.

—The "clock speed," or memory cycle speed, of most Commodore machines is roughly 1 microsecond—one millionth of a second. The precise number varies from one machine to another. and also varies between North America and other regions.

—Most instructions run at the fastest imaginable speed. Count the memory cycles, and that's how fast the instruction will execute. For example, LDA #$DD will need two memory cycles just to get the instruction—and that's how fast it runs LDA $D5DD, X will usually take four memory cycles. three to get the instruction, and one to fetch the data from page 5 Exceptions· no instruction runs in less than two cycles; and shift/rotate instructions, INC/DEC, and JSR/RTS take longer than you might expect by this rule.

—Branches time differently, depending on whether the branch is taken (three cycles) or not taken (two cycles)

—When a page boundary is crossed, the computer needs an extra cycle to do the arithmetic. If the program branches from $DFE4 to $1D23, there will be an extra cycle; if we LDA $24E7,Y, there will be an extra cycle if Y contains a value of $19 or greater.

Detailed timing values can be obtained from most tables of instructions.

Let's take a simple routine and estimate its timing. The following program logically ANDs the contents of 1DD locations from $17ED to $1844:

```
033C    LDX    #$DD
033E    LDA    #$DD
0340    AND    $17ED,X
0343    INX
```

```
0345   CPX   #$64
0347   BCC   $0340
0349   RTS
```

We may work out timing as follows·

LDX #$00—executed once·	2
LDA #$00—executed once·	2
AND $17E0,X: 32 times at 4 cycles	128
68 times at 5 cycles (page cross)	340
INX—100 times at 2 cycles	200
CPX #$64—100 times at 2 cycles.	200
BCC—99 times at 3 cycles	297
1 time at 2 cycles (no branch)	2
RTS—6 cycles.	6

Total time: 1171 cycles, or slightly over one thousandth of a second. We might add 10 percent to allow for the effects of interrupt; and since this is a subroutine, we could also add the extra six cycles needed to perform the JSR.

Where timing is critical, the interrupt could be locked out with SEI Be careful: it's seldom necessary, and is potentially dangerous.

Input and Output

We know that calling the kernal routine CHROUT at $FFD2 will send an ASCII character to the screen. *We may also redirect output to any logical file.*

We have seen that we may obtain input from the keyboard buffer into the A register by calling kernal routine GETIN at $FFE4. *We may also redirect the input so that we draw information from any logical file.*

The same commands—$FFD2 and $FFE4—still perform the input and output. But we "switch" either of them to connect to a chosen device—or more accurately, a chosen logical file. The file must be open; we may switch to the file, and then switch back to normal I/O as we wish.

Switching Output

We use subroutine CHKOUT at address $FFC9 to switch output to a logical file. When we want to restore output to the screen, we call subroutine CLRCHN at $FFCC. This is not the same as an OPEN and

Figure 8.1

C L O S E—we simply connect to the file and disconnect, and we can do this as many times as we want.

Subroutine:	CHKOUT
Address	$FFC9
Action.	Switches the output path (used by CHROUT, $FFD2) so that output is directed to the logical file specified in the X register. The logical file must previously have been opened.

The character subsequently sent by $FFD2 is usually ASCII (or PET ASCII). When sent to the printer, special characters—text/graphics, width—will be honored in the usual way. Similarly, disk commands can be transmitted over secondary address 15 if desired; a logical "command channel" file must be previously opened.

Registers: Registers A and X will be changed during the CHKOUT call. Be sure to save any sensitive data in these registers before calling CHKOUT.

Status: Status flags may be changed. In VIC and Commodore 64, the C (carry) flag indicates some type of problem with connecting to the output channel.

To switch output to logical file 1, we would need to follow these steps:

SCREEN

T
S

FFC9)
HE
VITCH

d we can do

, $FFD2)
ecified in the
have been

Cll (or PET
xt/graphics,
mands can
gical "com-

CHKOUT
fore calling

ore 64, the
ting to the

ese steps:

1 Load a value of 1 into X (LDX #$01)

2. JSR to address $FFC9.

Once the output is switched, we may send as many characters as we wish using subroutine $FFD2. Eventually, we must disconnect from the logical file and return to our default output, the screen. We do this by calling subroutine CLRCHN at address $FFCC.

Subroutine.	CLRCHN
Address:	$FFCC
Action:	Disconnects input and output from any logical files and restores them to the "default" input and output channels, keyboard and screen. The logical files are not closed, and may be reconnected at a later time.

Registers: Registers A and X will be changed during the CLRCHN call. Be sure to save any sensitive data in these registers.

Status: Status flags may be changed. In VIC and Commodore 64, the C (carry) flag indicates some type of problem with output.

The *logical file* concept is important. I may send to any destination—cassette, printer, disk, or screen—without knowing which device is involved. I send the characters on their way and the operating system sees that they are delivered wherever they need to go.

This simplifies the machine language programmer's job. It's a simple task to send the characters to some logical channel; the programmer does not need to take special coding action depending on which device is involved.

Output Example

If we wanted to print the message HI on the printer, we might code as follows.

First, we'll open the printer channel in BASIC. Let's use logical file number 1:

```
100 OPEN 1,4
110 SYS 828
120 CLOSE 1
```

If you don't have a printer, you may open the file to cassette (OPEN 1,1,2) or to disk (OPEN 1,8,3, "0:DEMO,S,W"). The machine language program won't care: it will send to logical file number 1 no matter what it is; it might even be the screen (OPEN 1,3). Let's write the coding:

```
.A  033C  LDX  #$01
.A  033E  JSR  $FFC9
```

Now the output is connected to logical file 1. Let's say HI:

```
.A  0341  LDA  #$48
.A  0343  JSR  $FFD2
.A  0346  LDA  #$49
.A  0348  JSR  $FFD2
.A  034B  LDA  #$0D
.A  034D  JSR  $FFD2
.A  0350  JSR  $FFCC
.A  0353  RTS
```

Don't forget to send the RETURN—the printer needs it. After the machine language program says HI, the program will return to BASIC and close the file. Notice that the machine language program doesn't care what it's saying HI to ... it sends the data to logical file 1.

Switching Input

We use subroutine CHKIN at address $FFC6 to switch input so as to draw data from a logical file. When we want to restore input from the keyboard, we call subroutine CLRCHN at $FFCC. Again, this is not the same as an OPEN and CLOSE—we simply connect to the file and disconnect, and we can do this as many times as we want.

Inp

Subroutine	CHKIN
Address	$FFC6
Action.	Switches the input path (used by GET, $FFE4) so that input is taken from the logical file specified in the X register The logical file must previously have been opened

The character subsequently obtained by $FFE4 into the A register is usually ASCII (or PET ASCII) A binary zero received from a file usually represents exactly that: an input character whose value is CHR$(0), this is different from keyboard GET where a binary zero means "no key pressed " When accessing a file, ST (address $90 for VIC and Commodore 64, $96 for most PET/CBM) is used for its usual functions of signalling end-of-file or error. Similarly, disk status information can be received over secondary address 15 if desired, a logical "command channel" file must be previously opened.

Registers: Registers A and X will be changed during the CHKIN call Be sure to save any sensitive data in these registers before calling CHKIN.

Status. Status flags may be changed. In VIC and Commodore 64, the C (carry) flag indicates some type of problem with connecting to the input channel.

To switch input to logical file 1, we would need to follow these steps:

—Load a value of 1 into X (LDX #$01)

—JSR to address $FFC6

Once the input is switched, we may obtain as many characters as we wish using subroutine $FFE4. Eventually, we must disconnect from the logical file and return to our default input—the keyboard. We do this by calling subroutine CLRCHN at address $FFCC. This is the same subroutine that disconnects output from a logical file.

Input Example

We can write a program to read an input file from disk or cassette. First, let's write the file. We open the file according to its type:

Disk   OPEN 1,8,3,"0:DEMO,S,W"

Cassette: OPEN 1,1,1

This may be done with a direct statement. Now let's write a few things to the file:

```
PRINT#1, "HELLO THIS IS A TEST"
PRINT#1,"THIS IS THE LAST LINE"
CLOSE 1
```

If we have typed in the above statements correctly, we should have a completed sequential file written on cassette or disk. Before writing the machine language input program, let's examine how we might read the file back in BASIC:

Disk·   100 OPEN 1,8,3,"DEMO"

Cassette: 100 OPEN 1

                110 INPUT #1,X$

                120 PRINT X$

                130 IF ST=0 GOTO 110

                140 CLOSE 1

We might alternatively have written lines 110 and 120 as

```
110 GET #1,X$
120 PRINT X$;
```

This more closely approximates the logic flow of our machine language program, since it will get the characters one at a time. If you are unsure about the role of ST, read up on it. We will use the same variable (at its address of $90 or $96) to do exactly the same thing in machine language.

Type NEW and enter the following program:

```
Disk        100 OPEN 1,8,3,"DEMO"
Cassette:    100 OPEN 1
             110 SYS 828
             120 CLOSE 1
```

We will read the file and copy it to the screen entirely in machine language. Let's start coding at $033C :

```
.A   033C   LDX   #$01
.A   033E   JSR   $FFC6
```

Now the input is connected to logical file 1. Let's get information from it and put it on the screen:

```
.A   0341   JSR   $FFE4
.A   0344   JSR   $FFD2
```

We must check ST as we would in BASIC. ST might be at either of two addresses, depending on the system:

```
VIC, Commodore 64.   .A 0347 LDA $90
CBM/PET              .A 0347 LDA $96
```

If ST is zero, there is more to come from the file; we may go back. If ST is nonzero, there could be an error or we may be at the end of the file. In either case, we don't want to read more from the file.

```
.A   0349   BEQ   $0341
.A   034B   JSR   $FFCC
.A   034E   RTS
```

Check it and try it. The file is delivered to the screen quickly.

A File Transfer Program

Let's write a program to transfer a sequential file from any common device to any other. BASIC will sort out which files to handle; once the files are opened, machine language will take from and deliver to the appropriate logical devices as desired

It's not a good idea to switch input and output at the same time—in other words, to call both $FFC6 and $FFC9 without canceling either via $FFCC. The kernal doesn't mind, but it confuses the peripheral devices, which expect to have exclusive occupancy of the data bus to the computer. So we'll follow the pattern: switching on, sending or receiving, switching off, and then going to the other device.

One more thing. ST tells us the status of the *last device handled*. Consider: if we input a character, then output a character, and then check the value of ST, we have a problem. ST will not tell us about the input, since the last device handled was output; thus, we won't know if we are at the end of the file or not. In machine language, as in BASIC, we must code carefully to solve this problem.

Here comes BASIC:

```
100 PRINT "FILE TRANSFER"
110 INPUT "INPUT FROM (DISK, TAPE)";A$
120 IF LEFT$(A$,1)="T" THEN OPEN 1:GOTO 160
130 IF LEFT$(A$,1)<> "D" GOTO 110
140 INPUT "DISK FILE NAME";N$
150 OPEN 1,8,3,N$
160 INPUT "TO (DISK, TAPE, SCREEN)";B$
170 IF LEFT$(B$,1)="S" THEN OPEN 2,3:GOTO 240
180 IF LEFT$(B$,1)="D" GOTO 210
190 IF LEFT$(B$,1) <> "T" GOTO 160
200 IF LEFT$(A$,1)="T" GOTO 160
210 INPUT "OUTPUT FILE NAME";F$
220 IF LEFT$(B$,1)="D"
THEN OPEN 2,8,4,"0:"+N$+",S,W"
230 IF LEFT$(B$,1)="T" THEN OPEN 2,1,1,N$
240 SYS xxxx
250 CLOSE 2:CLOSE 1
```

We'll work this out for the Commodore 64 computer; you can adjust it for PET/CBM or VIC-20. The above BASIC program should not take up more than 511 bytes; on a standard Commodore 64, that means that we'll have clear space for our machine language program starting at $0A00 (decimal 2560). We'll move the start-of-variables along, of course, so that our machine language program won't be disturbed by them.

When we first type line 240, we won't know what SYS address to use. After the program is typed in (with SYS xxxx at line 240), we can easily confirm that the machine language can start at $0A00 by checking the start-of-variables pointer. We go back and change 240 to SYS 2560; now we're ready to put in the machine language code:

```
. A   0A00   LDX   #$01
. A   0A02   JSR   $FFC6
. A   0A05   JSR   $FFE4
```

By this time, we have a character in the A register from the input source.

We also have a value in ST, telling us if this is the last character. Let's examine the ST problem: we must check its value now, since ST will be changed after we do the output. But we don't want to take any action based on ST yet; we must first send the character we have received. Let's check ST, and put the results of the check onto the stack:

```
.A  0A08  LDX  $90
.A  0A0A  PHP
```

If ST is zero, the Z flag will be set; we'll preserve this flag along with the others until we call it back from the stack. If you are adapting this program for the PET/CBM, don't forget that ST is at address $96 for your machine.

The next thing we want to do is to disconnect the input by calling $FFCC; but this will destroy the A register. How can we preserve this value? By transferring to another register, or by pushing A to the stack. Let's do that. There will now be two things on the stack.

```
.A 0A0B PHA
```

We are now free to disconnect from the input channel and connect to the output. Here we go:

```
.A  0A0C  JSR  $FFCC
.A  0A0F  LDX  #$02
.A  0A11  JSR  $FFC9
.A  0A14  PLA
```

The A register gets back the last thing saved to the stack, and that, of course, is our input character. We're ready to send it to the output device:

```
.A  0A15  JSR  $FFD2
.A  0A18  JSR  $FFCC
```

Now we may pick up on the condition of ST that we stacked away earlier. Here come the flags that we stored:

```
.A 0A1B PLP
```

If the Z flag is set, we want to go back and get another character. If not, we're finished and can return to BASIC, allowing BASIC to close the files for us:

```
.A  0A1C  BEQ  $0A00
.A  0A1E  RTS
```

Important: Before running this program, be sure to move the start-of-

8

Timing, Input/Output, and Conclusion

This chapter discusses:

- How to estimate the speed of your program
- Input and output from tape, disk, and printer
- Review of instructions
- Debugging
- Symbolic assemblers
- Where to go from here

Timing

For many applications, machine language programs seem to run instantaneously. The speed of the 650x is much greater than that of other devices, including the human user. The machine language program usually ends up waiting for something: waiting for the keyboard, waiting for the printer, waiting for the disk, or waiting for the human to read and react to information presented on the screen.

Occasionally, it may be important to get fairly precise timing for a machine language program. If so, the following rules of thumb may be kept in mind:

—All timing estimates are crude if the interrupt routines are still active. The effect of interrupt on timing can be crudely estimated by adding 10 percent to the running time.

—Remember to allow for loops. If an instruction within a loop is repeated ten times, its timing will need to be counted ten times.

—The "clock speed," or memory cycle speed, of most Commodore machines is roughly 1 microsecond—one millionth of a second. The precise number varies from one machine to another, and also varies between North America and other regions.

—Most instructions run at the fastest imaginable speed. Count the memory cycles, and that's how fast the instruction will execute. For example, LDA #$0D will need two memory cycles just to get the instruction—and that's how fast it runs LDA $0500,X will usually take four memory cycles. three to get the instruction, and one to fetch the data from page 5 Exceptions· no instruction runs in less than two cycles; and shift/rotate instructions, INC/DEC, and JSR/RTS take longer than you might expect by this rule.

—Branches time differently, depending on whether the branch is taken (three cycles) or not taken (two cycles)

—When a page boundary is crossed, the computer needs an extra cycle to do the arithmetic. If the program branches from $0FE4 to $1023, there will be an extra cycle; if we LDA $24E7,Y, there will be an extra cycle if Y contains a value of $19 or greater.

Detailed timing values can be obtained from most tables of instructions.

Let's take a simple routine and estimate its timing. The following program logically ANDs the contents of 100 locations from $17E0 to $1844:

```
033C   LDX   #$00
033E   LDA   #$00
0340   AND   $17E0,X
0343   INX
```

```
0345   CPX   #$64
0347   BCC   $0340
0349   RTS
```

We may work out timing as follows·

LDX #$00—executed once·	2
LDA #$00—executed once·	2
AND $17E0,X: 32 times at 4 cycles	128
68 times at 5 cycles (page cross)	340
INX—100 times at 2 cycles	200
CPX #$64—100 times at 2 cycles.	200
BCC—99 times at 3 cycles	297
1 time at 2 cycles (no branch)	2
RTS—6 cycles.	6

Total time: 1171 cycles, or slightly over one thousandth of a second. We might add 10 percent to allow for the effects of interrupt; and since this is a subroutine, we could also add the extra six cycles needed to perform the JSR.

Where timing is critical, the interrupt could be locked out with SEI Be careful: it's seldom necessary, and is potentially dangerous.

Input and Output

We know that calling the kernal routine CHROUT at $FFD2 will send an ASCII character to the screen. *We may also redirect output to any logical file.*

We have seen that we may obtain input from the keyboard buffer into the A register by calling kernal routine GETIN at $FFE4. *We may also redirect the input so that we draw information from any logical file.*

The same commands—$FFD2 and $FFE4—still perform the input and output. But we "switch" either of them to connect to a chosen device—or more accurately, a chosen logical file. The file must be open; we may switch to the file, and then switch back to normal I/O as we wish.

Switching Output

We use subroutine CHKOUT at address $FFC9 to switch output to a logical file. When we want to restore output to the screen, we call subroutine CLRCHN at $FFCC. This is not the same as an OPEN and

KEYBOARD

INPUT

PROGRAM

SCREEN

INPUT
DEVICES

DEVICES

CHKIN ($FFC6)
SETS THE
INPUT SWITCH

CHKOUT ($FFC9)
SETS THE
OUTPUT SWITCH

CLRCHN ($FFCC)
RESTORES BOTH
SWITCHES TO "NORMAL"

Figure 8.1

CLOSE—we simply connect to the file and disconnect, and we can do this as many times as we want.

Subroutine:	CHKOUT
Address	$FFC9
Action.	Switches the output path (used by CHROUT, $FFD2) so that output is directed to the logical file specified in the X register. The logical file must previously have been opened.

The character subsequently sent by $FFD2 is usually ASCII (or PET ASCII). When sent to the printer, special characters—text/graphics, width—will be honored in the usual way. Similarly, disk commands can be transmitted over secondary address 15 if desired; a logical "command channel" file must be previously opened.

Registers: Registers A and X will be changed during the CHKOUT call. Be sure to save any sensitive data in these registers before calling CHKOUT.

Status: Status flags may be changed. In VIC and Commodore 64, the C (carry) flag indicates some type of problem with connecting to the output channel.

To switch output to logical file 1, we would need to follow these steps:

1 Load a value of 1 into X (LDX #$01)

2. JSR to address $FFC9.

Once the output is switched, we may send as many characters as we wish using subroutine $FFD2. Eventually, we must disconnect from the logical file and return to our default output, the screen. We do this by calling subroutine CLRCHN at address $FFCC.

Subroutine.   CLRCHN
Address: $FFCC
Action: Disconnects input and output from any logical files and restores them to the "default" input and output channels, keyboard and screen. The logical files are not closed, and may be reconnected at a later time.

Registers: Registers A and X will be changed during the CLRCHN call. Be sure to save any sensitive data in these registers.

Status: Status flags may be changed. In VIC and Commodore 64, the C (carry) flag indicates some type of problem with output.

The *logical file* concept is important. I may send to any destination—cassette, printer, disk, or screen—without knowing which device is involved. I send the characters on their way and the operating system sees that they are delivered wherever they need to go.

This simplifies the machine language programmer's job. It's a simple task to send the characters to some logical channel; the programmer does not need to take special coding action depending on which device is involved.

Output Example

If we wanted to print the message HI on the printer, we might code as follows.

First, we'll open the printer channel in BASIC. Let's use logical file number 1:

```
100 OPEN 1,4
110 SYS 828
120 CLOSE 1
```

If you don't have a printer, you may open the file to cassette (OPEN 1,1,2) or to disk (OPEN 1,8,3, "0:DEMO,S,W"). The machine language program won't care: it will send to logical file number 1 no matter what it is; it might even be the screen (OPEN 1,3). Let's write the coding:

```
.A  033C  LDX  #$01
.A  033E  JSR  $FFC9
```

Now the output is connected to logical file 1. Let's say HI:

```
.A  0341  LDA  #$48
.A  0343  JSR  $FFD2
.A  0346  LDA  #$49
.A  0348  JSR  $FFD2
.A  034B  LDA  #$0D
.A  034D  JSR  $FFD2
.A  0350  JSR  $FFCC
.A  0353  RTS
```

Don't forget to send the RETURN—the printer needs it. After the machine language program says HI, the program will return to BASIC and close the file. Notice that the machine language program doesn't care what it's saying HI to . . . it sends the data to logical file 1.

Switching Input

We use subroutine CHKIN at address $FFC6 to switch input so as to draw data from a logical file. When we want to restore input from the keyboard, we call subroutine CLRCHN at $FFCC. Again, this is not the same as an OPEN and CLOSE—we simply connect to the file and disconnect, and we can do this as many times as we want.

Subroutine	CHKIN
Address	$FFC6
Action.	Switches the input path (used by GET, $FFE4) so that input is taken from the logical file specified in the X register. The logical file must previously have been opened.

The character subsequently obtained by $FFE4 into the A register is usually ASCII (or PET ASCII). A binary zero received from a file usually represents exactly that: an input character whose value is CHR$(0), this is different from keyboard GET where a binary zero means "no key pressed." When accessing a file, ST (address $90 for VIC and Commodore 64, $96 for most PET/CBM) is used for its usual functions of signalling end-of-file or error. Similarly, disk status information can be received over secondary address 15 if desired, a logical "command channel" file must be previously opened.

Registers: Registers A and X will be changed during the CHKIN call. Be sure to save any sensitive data in these registers before calling CHKIN.

Status. Status flags may be changed. In VIC and Commodore 64, the C (carry) flag indicates some type of problem with connecting to the input channel.

To switch input to logical file 1, we would need to follow these steps:

—Load a value of 1 into X (LDX #$01)

—JSR to address $FFC6

Once the input is switched, we may obtain as many characters as we wish using subroutine $FFE4. Eventually, we must disconnect from the logical file and return to our default input—the keyboard. We do this by calling subroutine CLRCHN at address $FFCC. This is the same subroutine that disconnects output from a logical file.

Input Example

We can write a program to read an input file from disk or cassette. First, let's write the file. We open the file according to its type:

Disk   OPEN 1,8,3,"0:DEMO,S,W"
Cassette: OPEN 1,1,1

This may be done with a direct statement. Now let's write a few things to the file:

```
PRINT#1, "HELLO THIS IS A TEST"
PRINT#1,"THIS IS THE LAST LINE"
CLOSE 1
```
If we have typed in the above statements correctly, we should have a completed sequential file written on cassette or disk. Before writing the machine language input program, let's examine how we might read the file back in BASIC:

Disk·   100 OPEN 1,8,3,"DEMO"
Cassette: 100 OPEN 1
            110 INPUT #1,X$
            120 PRINT X$
            130 IF ST=0 GOTO 110
            140 CLOSE 1

We might alternatively have written lines 110 and 120 as

```
110 GET #1,X$
120 PRINT X$;
```

This more closely approximates the logic flow of our machine language program, since it will get the characters one at a time. If you are unsure about the role of ST, read up on it. We will use the same variable (at its address of $90 or $96) to do exactly the same thing in machine language.

Type NEW and enter the following program:

```
Disk          100 OPEN 1,8,3,"DEMO"
Cassette:     100 OPEN 1
              110 SYS 828
              120 CLOSE 1
```

We will read the file and copy it to the screen entirely in machine language. Let's start coding at $033C:

```
.A  033C  LDX  #$01
.A  033E  JSR  $FFC6
```

Now the input is connected to logical file 1. Let's get information from it and put it on the screen:

```
.A  0341  JSR  $FFE4
.A  0344  JSR  $FFD2
```

We must check ST as we would in BASIC. ST might be at either of two addresses, depending on the system:

```
VIC, Commodore 64.   .A 0347 LDA $90
CBM/PET              .A 0347 LDA $96
```

If ST is zero, there is more to come from the file; we may go back. If ST is nonzero, there could be an error or we may be at the end of the file. In either case, we don't want to read more from the file.

```
.A  0349  BEQ  $0341
.A  034B  JSR  $FFCC
.A  034E  RTS
```

Check it and try it. The file is delivered to the screen quickly.

A File Transfer Program

Let's write a program to transfer a sequential file from any common device to any other. BASIC will sort out which files to handle; once the files are opened, machine language will take from and deliver to the appropriate logical devices as desired

It's not a good idea to switch input and output at the same time—in other words, to call both $FFC6 and $FFC9 without canceling either via $FFCC. The kernal doesn't mind, but it confuses the peripheral devices, which expect to have exclusive occupancy of the data bus to the computer. So we'll follow the pattern: switching on, sending or receiving, switching off, and then going to the other device.

One more thing. ST tells us the status of the *last device handled*. Consider: if we input a character, then output a character, and then check the value of ST, we have a problem. ST will not tell us about the input, since the last device handled was output; thus, we won't know if we are at the end of the file or not. In machine language, as in BASIC, we must code carefully to solve this problem.

Here comes BASIC:

```
100 PRINT "FILE TRANSFER"
110 INPUT "INPUT FROM (DISK, TAPE)";A$
120 IF LEFT$(A$,1)="T" THEN OPEN 1:GOTO 160
130 IF LEFT$(A$,1)<> "D" GOTO 110
140 INPUT "DISK FILE NAME";N$
150 OPEN 1,8,3,N$
160 INPUT "TO (DISK, TAPE, SCREEN)";B$
170 IF LEFT$(B$,1)="S" THEN OPEN 2,3:GOTO 240
180 IF LEFT$(B$,1)="D" GOTO 210
190 IF LEFT$(B$,1) <> "T" GOTO 160
200 IF LEFT$(A$,1)="T" GOTO 160
210 INPUT "OUTPUT FILE NAME";F$
220 IF LEFT$(B$,1)="D"
THEN OPEN 2,8,4,"0:"+N$+",S,W"
230 IF LEFT$(B$,1)="T" THEN OPEN 2,1,1,N$
240 SYS xxxx
250 CLOSE 2:CLOSE 1
```

We'll work this out for the Commodore 64 computer; you can adjust it for PET/CBM or VIC-20. The above BASIC program should not take up more than 511 bytes; on a standard Commodore 64, that means that we'll have clear space for our machine language program starting at $0A00 (decimal 2560). We'll move the start-of-variables along, of course, so that our machine language program won't be disturbed by them.

When we first type line 240, we won't know what SYS address to use. After the program is typed in (with SYS xxxx at line 240), we can easily confirm that the machine language can start at $0A00 by checking the start-of-variables pointer. We go back and change 240 to SYS 2560; now we're ready to put in the machine language code:

```
.A  0A00  LDX  #$01
.A  0A02  JSR  $FFC6
.A  0A05  JSR  $FFE4
```

By this time, we have a character in the A register from the input source.

We also have a value in ST, telling us if this is the last character. Let's examine the ST problem: we must check its value now, since ST will be changed after we do the output. But we don't want to take any action based on ST yet; we must first send the character we have received. Let's check ST, and put the results of the check onto the stack:

```
.A  0A08  LDX  $90
.A  0A0A  PHP
```

If ST is zero, the Z flag will be set; we'll preserve this flag along with the others until we call it back from the stack. If you are adapting this program for the PET/CBM, don't forget that ST is at address $96 for your machine.

The next thing we want to do is to disconnect the input by calling $FFCC; but this will destroy the A register. How can we preserve this value? By transferring to another register, or by pushing A to the stack. Let's do that. There will now be two things on the stack.

```
.A 0A0B PHA
```

We are now free to disconnect from the input channel and connect to the output. Here we go:

```
.A  0A0C  JSR  $FFCC
.A  0A0F  LDX  #$02
.A  0A11  JSR  $FFC9
.A  0A14  PLA
```

The A register gets back the last thing saved to the stack, and that, of course, is our input character. We're ready to send it to the output device:

```
.A  0A15  JSR  $FFD2
.A  0A18  JSR  $FFCC
```

Now we may pick up on the condition of ST that we stacked away earlier. Here come the flags that we stored:

```
.A 0A1B PLP
```

If the Z flag is set, we want to go back and get another character. If not, we're finished and can return to BASIC, allowing BASIC to close the files for us:

```
.A  0A1C  BEQ  $0A00
.A  0A1E  RTS
```

Important: Before running this program, be sure to move the start-of-

variables pointer ($□□2D/$□□2E) so that it points at address $□A1F; otherwise, the BASIC variables will destroy this program.

Review: The Instruction Set

We started with the load, save and compare for the three data registers:

```
LDA   LDX   LDY
STA   STX   STY
CMP   CPX   CPY
```

The instructions are almost identical in action, although only the A register has indirect, indexed addressing modes. We continued with the logical and arithmetic routines that apply only to A·

```
AND   ORA   EOR   ADC   SBC
```

Arithmetic also includes the shift and rotate instructions, which may be used on the A register or directly upon memory:

```
ASL   ROL   LSR   ROR
```

Memory may also be directly modified by the increment and decrement instructions, which have related instructions that operate on X and Y:

```
INC   DEC
INX   DEX
INY   DEY
```

We may transfer control by means of branch instructions, which are all conditional:

```
BEQ   BCS   BMI   BVS
BNE   BCC   BPL   BVC
```

The branch instructions can make only short "hops"; the jump instruction is unconditional:

```
JMP
```

Subroutines are called with the jump-subroutine, and returned with return-from-subroutine; we may also return from interrupts:

```
JSR   RTS   RTI
```

We may modify any of several flags with the appropriate set or clear command. Some of the flags control internal processor operation: for example, the I (interrupt disable) flag locks out the interrupt; the D (decimal mode) affects the way the ADC and SBC work with numbers.

```
SEC        SEI  SED
CLC   CLV  CLI  CLD
```

We may transfer information between the A register and X or Y; and for checking or setting the stack location, we may move the stack pointer to X, or X to the stack pointer. The latter is a powerful command, so use it with care.

```
TAX   TAY  TSX
TXA   TYA  TXS
```

We may push or pull information from the stack:

```
PHA   PHP
PLA   PLP
```

There's a special test, used mostly for checking I A chips:

```
BIT
```

The BIT test is used only for specific locations: no indexing is allowed. The high bit from the location being tested is transferred straight to the N flag. The next highest bit (bit 6) goes straight to the V flag. Finally, the Z flag is set according to whether the location has any bits set that match bits set in the A register. Thus, we can check a location with BIT $ . . . . followed by BMI to test the high bit, or BVS to test bit 6, or BNE to test any selected bit or group of bits. It's a rather specialized instruction, but useful for testing input/output ports.

Finally, the instruction that does nothing, and the BRK instruction that causes a "false interrupt," usually taking us to the monitor:

```
NOP   BRK
```

That's the whole set. With these instructions, you can write programs to make the computer do whatever you choose.

Debugging

When a program has been written. the next step is to look for any possible errors, or *bugs*. The process of searching for and systematically eliminating these bugs is called *debugging*.

Most programs are made up of sections, each of which has a clear task to perform. When a program misbehaves, it may be easy to go to the area of the bug, since you can see which parts of the program are working and where things start to go wrong.

In case of doubt, you may insert *breakpoints* into your program. Replace selected instructions with the instruction BRK, this may be done by replacing the instructions' op codes with the value ☐☐. Run the program; when it reaches the first breakpoint, it will stop and the machine language monitor will become active. Examine the registers carefully to see whether they contain the values expected. Display memory locations that the program should have written; the contents will tell you whether the program has been doing its job correctly.

When you have confirmed that the program is behaving correctly up to the breakpoint, replace the BRK command at that point with the original op code. Command .G to that address, and the program will continue to the next breakpoint. If it helps your investigation, you may even change memory or registers before continuing program execution.

If you carried this procedure to the extreme, you might stop your program after every instruction. It would take time, but you would certainly track down everything the program did.

The best debugging takes place at the time you write the program. Write sensibly, not "super cleverly." If you fear getting caught in an endless loop, insert a stop key test (JSR $FFE1) so that you'll still have control of the computer.

Get to know your machine language monitor. The monitor uses a number of locations in memory; you'll have trouble debugging a program if it uses the same storage addresses as does your program. Every time you try to check the contents of a memory location to see what your program has done, you'll see the monitor working values instead—and that would be misleading and annoying.

Symbolic Assemblers

Throughout these exercises, we have used small, "nonsymbolic" assemblers such as would be found within a machine language monitor. These are good for beginners; they parallel the machine code quite closely and allow you to keep the working machine clearly in focus.

As you write bigger and better programs, these small assemblers will be less convenient. Forward branches and subroutines we have not yet written make it necessary for us to "guess" at the address and fix up our guess later. There is the possible danger that an address will be typed in wrongly ($☐∃45 instead of $☐∃54), causing the program to fail.

To help us write more ambitious programs, we may wish to turn to com-

mercially available assembler systems that allow *labels* or *symbolic addresses*. If we wish to write code to call a subroutine to input numbers—we might not have written this subroutine yet—we can code JSR NUMIN. When we write the subroutine, we'll put the identifying label NUMIN at the start. As your program is assembled, the proper address of NUMIN is determined, and this address will be inserted as needed.

It saves work and helps guard against errors. But symbolic assemblers allow a more powerful capability: they help documentation and allow program updating.

Your assembly may be listed to the printer. This allows you to examine and annotate the program, and file the details away for later reference. The assembler allows you to include *comments*, which improve the readability of the listing but don't affect the machine language program.

The *source program* you have written may be saved and used again later. If you find it is necessary to change the program, bring back the source code from cassette or disk, make the changes, and reassemble. In this way, programs can be easily corrected or updated.

Where To Go From Here

Almost anywhere. Up to this point, we've been building confidence: trying to give you a feel as to how the pieces work. Now, the real fun—the creative programming—is up to you.

Users have varying objectives. You may want to do mathematical operations. You may want to interact upon BASIC programs—analyzing, searching, renumbering. Whatever suits you. Your interest area may be music, graphics, or animation. Machine language will open the door to all of these; its amazing speed makes spectacular effects possible You may plan to go into hardware and interface new devices to your computer; an understanding of machine language, and IA chips in particular, will be useful. The possibilities are endless.

Even if you have no immediate plans to write new programs in machine language, you will have gained an insight into the workings of your machine. Everything that the machine does—BASIC, kernal, everything—is either hardware or machine language.

With the elementary concepts we have introduced here, you will be able to go deeper into more advanced texts Many programming books deal with the abstract 650x chip. That's hard for the beginner; it's difficult to see how the instructions fit within the architecture of a real machine, or

how the programs can actually be placed within the computer. By now, you should be able to take a piece of abstract coding and fit it into your system.

Many things start to happen at once when you take your first steps in machine language programming. You must learn how to use the monitor. You must learn a good deal about how your machine is designed. And you must learn how to fit the pieces together. It takes a while to adapt to the "information shock"—but things start to fit together. Eventually, you'll have a stronger and sounder view of the whole computer: hardware, software, languages, and usage.

What You Have Learned

—Machine language programs can have run times estimated fairly accurately In many cases, however, machine language is so fast that detailed speed calculations are not needed.

—We can handle input from devices other than the keyboard by switching the identity of the designated input device If an input channel has been opened as a file, we may connect to it with J S R $FFC6 and disconnect with J S R $FFCC

—We can handle output to devices other than the screen by switching the identity of the designated output device If an output channel has been opened as a file, we may connect to it with J S R $FFC9 and disconnect with J S R $FFCC

—Once input or output has been switched, we may receive in the usual way with the subroutine at $FFE4, or send in the usual way with the subroutine at $FFD2.

—Be careful not to confuse *connecting to a channel* with *opening a file* In a typical program, we open a file only once, but we may connect to it and disconnect from it hundreds of times as we read or write data

—You have met all the instructions of the 650x microprocessor There are enough for versatility, but not so many that you can't keep track of them all. You have made a worthwhile start in the art and science of machine language programming.

Questions and Projects

Write a program to read a sequential file and count the number of times the letter "A" (hex 41) appears in the file. Use a BASIC PEEK to print the value. You may assume that "A" will not appear more than 255 times.

Rewrite the above to count the number of occurrences of the RETURN character ($0D) in a sequential file. Allow for up to 65535 appearances. Can you attach a meaning to this count?

Write a program to print HAPPY NEW YEAR to the printer ten times.

If you own a disk system, you know that you can scratch a program named JUNK by using the sequence:

OPEN 15,8,15:PRINT#15,"S0:JUNK". Convert the PRINT# statement to machine language and write a program to scratch JUNK. Careful: don't scratch a program that you will need.

Write a "typewriter" program to read a line of text from the keyboard and then transfer it to the printer. It will be a more useful program if you show what is being typed on the screen and if you write extra code to honor the DELETE key.

A

The 6502/ 6510/6509/ 7501 Instruction Set

The four chips differ only in their use of addresses 0 and 1:

On the 6502, the addresses are normal memory.

On the 6510 and 7501, address 0 is a directional register and address 1 is an input/output register, used for such things as cassette tape and memory control.

On the 6509, address 0 is used to switch program execution to a new memory bank; address 1 is used to switch the memory bank accessed by the two instructions LDA (..), Y and STA (..), Y

Addressing Modes

Accumulator Addressing—This form of addressing is represented with a one byte instruction, implying an operation on the accumulator.

Immediate Addressing—In immediate addressing, the operand is contained in the second byte of the instruction, with no further memory addressing required.

Absolute Addressing—In absolute addressing, the second byte of the instruction specifies the eight low order bits of the effective address while the third byte specifies the eight high order bits. Thus, the absolute addressing mode allows access to the entire 65K bytes of addressable memory.

Zero Page Addressing—The zero page instructions allow for shorter code and execution times by only fetching the second byte of the instruction and assuming a zero high address byte. Careful use of the zero page can result in significant increase in code efficiency.

Indexed Zero Page Addressing—(X, Y indexing)—This form of addressing is used in conjunction with the index register and is referred to as "Zero Page, X" or "Zero Page, Y." The effective address is calculated by adding the second byte to the contents of the index register. Since this is a form of "Zero Page" addressing, the content of the second byte references a location in page zero. Additionally, due to the "Zero Page" addressing nature of this mode, no carry is added to the high order eight bits of memory and crossing of page boundaries does not occur.

Indexed Absolute Addressing—(X. Y indexing)—This form of addressing is used in conjunction with X and Y index register and is referred to as "Absolute, X," and "Absolute, Y." The effective address is formed by adding the contents of X and Y to the address contained in the second and third bytes of the instruction. This mode allows the index register to contain the index or count value and the instruction to contain the base address. This type of indexing allows any location referencing and the index to modify multiple fields resulting in reduced coding and execution time.

Implied Addressing—In the implied addressing mode, the address containing the operand is implicitly stated in the operation code of the instruction.

Relative Addressing—Relative addressing is used only with branch instructions and establishes a destination for the conditional branch.

The second byte of the instruction becomes the operand which is an "Offset" added to the contents of the lower eight bits of the program counter

when the counter is set at the next instruction. The range of the offset is -128 to $+127$ bytes from the next instruction.

Indexed Indirect Addressing—In indexed indirect addressing (referred to as [Indirect, X]), the second byte of the instruction is added to the contents of the X index register, discarding the carry. The result of this addition points to a memory location on page zero whose contents are the low order eight bits of the effective address. The next memory location in page zero contains the high order eight bits of the effective address. Both memory locations specifying the high and low order bytes of the effective address must be in page zero.

Indirect Indexed Addressing—In indirect indexed addressing (referred to as [Indirect, Y]), the second byte of the instruction points to a memory location in page zero. The contents of this memory location are added to the contents of the Y index register, the result being the low order eight bits of the effective address. The carry from this addition is added to the contents of the next page zero memory location, the result being the high order eight bits of the effective address.

Absolute Indirect—The second byte of the instruction contains the low order eight bits of a memory location. The high order eight bits of that memory location is contained in the third byte of the instruction. The contents of the fully specified memory location are the low order byte of the effective address. The next memory location contains the high order byte of the effective address which is loaded into the sixteen bits of the program counter.

Instruction Set—Alphabetic Sequence

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break

BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive-OR" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift One Bit Right (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt

P

RTS   Return from Subroutine

SBC   Subtract Memory from Accumulator with Borrow

SEC   Set Carry Flag

SED   Set Decimal Mode

SEI   Set Interrupt Disable Status

STA   Store Accumulator in Memory

STX   Store Index X in Memory

STY   Store Index Y in Memory

TAX   Transfer Accumulator to Index X

TAY   Transfer Accumulator to Index Y

TSX   Transfer Stack Pointer to Index X

TXA   Transfer Index X to Accumulator

TXS   Transfer Index X to Stack Register

TYA   Transfer Index Y to Accumulator

Programming Model

Figure A.1

Figure A.2

	IMM 2	ZPAG 2	Z,X 2	(I,X) 2	(I),Y 2	ABS 3	A,X 3	A,Y 3
ORA	09	05	15	01	11	0D	1D	19
AND	29	25	35	21	31	2D	3D	39
EOR	49	45	55	41	51	4D	5D	59
ADC	69	65	75	61	71	6D	7D	79
STA		85	95	81	91	8D	9D	99
LDA	A9	A5	B5	A1	B1	AD	BD	B9
CMP	C9	C5	D5	C1	D1	CD	DD	D9
SBC	E9	E5	F5	E1	F1	ED	FD	F9

Op Code ends in −1, −5, −9, or −D

BPL	10	BMI	30
BVC	50	BVS	70
BCC	90	BCS	B0
BNE	D0	BEQ	F0

Branches −0

	IMM 2	ZPAG 2	Z,X 2	Z,Y 2	ABS 3	A,X 3	A,Y 3
ASL		06	16		0E	1E	
ROL		26	36		2E	3E	
LSR		46	56		4E	5E	
ROR		66	76		6E	7E	
STX		86		96	8E		
LDX	A2	A6		B6	AE		BE
DEC		C6	D6		CE	DE	
INC		E6	F6		EE	FE	

Op Code ends in −2, −6, or −E

	IMM 2	ZPAG 2	Z,X 2	ABS 3	A,X 3
BIT		24		2C	
STY		84	94	8C	
LDY	A0	A4	B4	AC	BC
CPY	C0	C4		CC	
CPX	E0	E4		EC	

Misc. −0, −4, −C

	ABS	(IND)
JSR	20	
JMP	4C	6C

Jumps

	0−	1−	2−	3−	4−	5−	6−	7−	8−	9−	A−	B−	C−	D−	E−	F−
−0	BRK				RTI		RTS									
−8	PHP	CLC	PLP	SEC	PHA	CLI	PLA	SEI	DEY	TYA	TAY	CLV	INY	CLD	INX	SED
−A	ASL-A		ROL-A		LSR-A		ROR-A		TXA	TXS	TAX	TSX	DEX		NOP	

Single-byte Op Codes −0, −8, −A

Mnemonic	Operation	Immediate OP	N	#	Absolute OP	N	#	Zero Page OP	N	#	Accum OP	N	#	Implied OP	N	#	(Ind) X OP	N	#	(Ind) Y OP	N	#	Z,Page,X OP	N	#	Abs X OP	N	#	Abs Y OP	N	#	Relative OP	N	#	Indirect OP	N	#	Z,Page,Y OP	N	#	N	Z	C	I	D	V
ADC	A + M + C → A (1)(4)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3										✓	✓	✓	-	-	✓
AND	A ∧ M → A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3										✓	✓	-	-	-	-
ASL	C ← [7]□[0] ← 0				0E	6	3	06	5	2	0A	2	1										16	6	2	1E	7	3													✓	✓	✓	-	-	-
BCC	Branch on C = 0 (2)																															90	2	2													
BCS	Branch on C = 1 (2)																															B0	2	2													
BEQ	Branch on Z = 1 (2)																															F0	2	2													
BIT	A ∧ M				2C	4	3	24	3	2																															M7✓	✓	-	-	-	M6
BMI	Branch on N = 1 (2)																															30	2	2													
BNE	Branch on Z = 0 (2)																															D0	2	2													
BPL	Branch on N = 0 (2)																															10	2	2													
BRK	(See Figure A-1)													00	7	1																														
BVC	Branch on V = 0 (2)																															50	2	2													
BVS	Branch on V = 1 (2)																															70	2	2													
CLC	0 → C													18	2	1																									-	-	0	-	-	-
CLD	0 → D													D8	2	1																									-	-	-	-	0	-
CLI	0 → I													58	2	1																									-	-	-	0	-	-
CLV	0 → V													B8	2	1																									-	-	-	-	-	0
CMP	A - M (1)	C9	2	2	CD	4	3	C5	3	2							C1	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3										✓	✓	✓	-	-	-
CPX	X - M	E0	2	2	EC	4	3	E4	3	2																															✓	✓	✓	-	-	-
CPY	Y - M	C0	2	2	CC	4	3	C4	3	2																															✓	✓	✓	-	-	-
DEC	M - 1 → M				CE	6	3	C6	5	2													D6	6	2	DE	7	3													✓	✓	-	-	-	-
DEX	X - 1 → X													CA	2	1																									✓	✓	-	-	-	-
DEY	Y - 1 → Y													88	2	1																									✓	✓	-	-	-	-
EOR	A ⊻ M → A (1)	49	2	2	4D	4	3	45	3	2							41	6	2	51	5	2	55	4	2	5D	4	3	59	4	3										✓	✓	-	-	-	-
INC	M + 1 → M				EE	6	3	E6	5	2													F6	6	2	FE	7	3													✓	✓	-	-	-	-
INX	X + 1 → X													E8	2	1																									✓	✓	-	-	-	-
INY	Y + 1 → Y													C8	2	1																									✓	✓	-	-	-	-
JMP	Jump to new loc				4C	3	3																												6C	5	3				-	-	-	-	-	-
JSR	Jump sub (See Figure A-2)				20	6	3																																		-	-	-	-	-	-
LDA	M → A (1)	A9	2	2	AD	4	3	A5	3	2							A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3										✓	✓	-	-	-	-

Figure A.3

Mnemonic	Operation	Immediate OP	N	#	Absolute OP	N	#	Zero Page OP	N	#	Accum OP	N	#	Implied OP	N	#	(Ind) X OP	N	#	(Ind) Y OP	N	#	Z, Page, X OP	N	#	Abs X OP	N	#	Abs Y OP	N	#	Relative OP	N	#	Indirect OP	N	#	Z, Page, Y OP	N	#	N	Z	C	I	D	V
LDX	M→X (1)	A2	2	2	AE	4	3	A6	3	2																BE	4	3							B6	4	2	✓	✓	–	–	–	–
LDY	M→Y (1)	A0	2	2	AC	4	3	A4	3	2													B4	4	2	BC	4	3											✓	✓	–	–	–	–
LSR	0→[7 0]→C				4E	6	3	46	5	2	4A	2	1										56	6	2	5E	7	3										0	✓	✓	–	–	–
NOP	NO OPERATION													EA	2	1																						–	–	–	–	–	–
ORA	AVM→A	09	2	2	0D	4	3	05	3	2							01	6	2	11	5	2	15	4	2	1D	4	3	19	4	3									✓	✓	–	–	–	–
PHA	A→Ms S – 1→S													48	3	1																						–	–	–	–	–	–
PHP	P→Ms S – 1→S													08	3	1																						–	–	–	–	–	–
PLA	S + 1→S MS→A													68	4	1																						✓	✓	–	–	–	–
PLP	S + 1→S MS→P													28	4	1																						(RESTORED)					
ROL	[7 0]←[C]				2E	6	3	26	5	2	2A	2	1										36	6	2	3E	7	3										✓	✓	✓	–	–	–
ROR	[C]→[7 0]				6E	6	3	66	5	2	6A	2	1										76	6	2	7E	7	3										✓	✓	✓	–	–	–
RTI	(See Figure A-1) RTRN INT													40	6	1																						(RESTORED)					
RTS	(See Figure A-1) RTRN SUB													60	6	1																						–	–	–	–	–	–
SBC	A – M – C̄→A (1)	E9	2	2	ED	4	3	E5	3	2							E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3									✓	✓	✓(3)	–	–	✓
SEC	1→C													38	2	1																						–	–	1	–	–	–
SED	1→D													F8	2	1																						–	–	–	–	1	–
SEI	1→I													78	2	1																						–	–	–	1	–	–
STA	A→M				8D	4	3	85	3	2							81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3									–	–	–	–	–	–
STX	X→M				8E	4	3	86	3	2																									96	4	2	–	–	–	–	–	–
STY	Y→M				8C	4	3	84	3	2													94	4	2													–	–	–	–	–	–
TAX	A→X													AA	2	1																						✓	✓	–	–	–	–
TAY	A→Y													A8	2	1																						✓	✓	–	–	–	–
TSX	S→X													BA	2	1																						✓	✓	–	–	–	–
TXA	X→A													8A	2	1																						✓	✓	–	–	–	–
TXS	X→S													9A	2	1																						–	–	–	–	–	–
TYA	Y→A													98	2	1																						✓	✓	–	–	–	–

(1) ADD 1 TO "N IF PAGE BOUNDARY IS CROSSED
(2) ADD 1 TO "N IF BRANCH OCCURS TO SAME PAGE
    ADD 2 TO N IF BRANCH OCCURS TO DIFFERENT PAGE
(3) CARRY NOT = BORROW
(4) IF IN DECIMAL MODE Z FLAG IS INVALID
    ACCUMULATOR MUST BE CHECKED FOR ZERO RESULT

X INDEX X
Y INDEX Y
A ACCUMULATOR
M MEMORY PER EFFECTIVE ADDRESS
MS MEMORY PER STACK POINTER

+ ADD
– SUBTRACT
∧ AND
∨ OR
∀ EXCLUSIVE OR

✓ MODIFIED
– NOT MODIFIED
M7 MEMORY BIT 7
M6 MEMORY BIT 6
N NO CYCLES
NO BYTES

Figure A.4

B

Some Characteristics of Commodore Machines

PET—Original ROM

The first PET. It can be recognized by the message seen at power up:

*** COMMODORE BASIC ***

using asterisks but with no identifying number after the word BASIC.

The original machine may be upgraded to Upgrade ROM by fitting a new set of ROM chips. This is a good idea, since the original logic cannot handle disk, does a poor job on cassette data files, has no built-in machine language monitor, and has a zero page architecture that differs significantly from all later PET/CBM's. The BASIC language on this unit is also limited; arrays may not contain over 256 elements, for example.

This early machine is becoming rare.

PET/CBM—Upgrade ROM

The first PET that can handle disk. It can be recognized by the message seen at power up:

COMMODORE BASIC

using the numbers sign (or octothorpe, if you like).

This is much cleaner logic than the previous machine. Its internal structure is similar to that of later PET/CBM units (the 4.0 machines), so that it has much in common with them.

It does not have specialized disk commands such as CATALOG, SCRATCH, or DLOAD (the 4.0 disk commands); but these are "convenience" commands and the Upgrade ROM unit can do everything that the later units do.

Upgrade ROM machines have a BASIC annoyance: under some circumstances, string variables need to be tidied up using a technique called "garbage collection." This takes place automatically when needed; but when it does, the machine will freeze and seemingly will be dead for a period that may last from a few seconds to as long as a half hour or more.

PET/CBM—4.0 ROM and 80 Characters

This class of machine has been a mainstay of the Commodore line for years. It may be recognized by the message seen at power up:

*** COMMODORE BASIC 4.0 ***

For the first time, a number appears in the message.

These machines are characterized by new BASIC disk commands (CATALOG, etc.) and elimination of garbage-collection delays. Their internal architecture, especially zero page, is quite similar to the previous Upgrade ROM computers.

Some time after the initial production of 40-column machines, 80-column machines were introduced, as well as a new 40-column version called the "fat 40." The later machines are distinguished by new screen/keyboard features, most noticeable of which is that the cursor movement keys repeat automatically.

Subsequently, two memory-expanded machines became available. The 8096 came fitted with 96K of RAM; the extra 64K was "bank switched" into memory as needed in blocks of 16K. The SuperPET, too, had an extra 64K of RAM that was bank switched in 4K blocks; it also came with an additional microprocessor (the 6809) used primarily for implementing high-level languages. Both the 8096 and the SuperPET may be used as conventional CBM 8032 computers; the extra memory may be ignored.

VIC-20

The VIC-20 was a new design concept for Commodore. Color, graphics, and sound were built into the computer The memory architecture changed radically. Zero-page locations were shifted significantly as compared to previous PET/CBM computers.

BASIC reverted to Upgrade ROM style—no special disk commands and potentially slow garbage collection. Other than that, BASIC was not trimmed. All the functions and features remained, and some attractive new screen editing features were added, such as automatic repeating keys.

The VIC comes with no machine language monitor, it's necessary to load one. The SYS command has a new attractive feature that allows registers A, X, and Y to be "preloaded" by POKEing values into addresses 780, 781, and 782. Location 783 could also be used to set the status register, but that's dangerous; unless it's done carefully, the decimal mode or interrupt disable flags might be set inadvertently.

The VIC-20 is somewhat vexing for machine language programming work. Depending on the amount of extra memory fitted (none, 3K, or 8K and over), the location of start-of-BASIC and of screen memory will vary.

Commodore 64

The Commodore 64 has much in common with the VIC-20. In particular, its zero page organization is almost identical to that of VIC. The Com-

modore 64 comes with a 6510 microprocessor; addresses 0 and 1 are reserved for "bank switching" of memory.

BASIC is identical to that of the VIC—no special disk commands and potentially slow garbage collection. There's no built-in machine language monitor, so one must be loaded. The SYS command, as with the VIC, allows preloading of registers A, X, and Y if desired.

The Commodore 64 has a more stable architecture than the VIC. BASIC starts in a consistent place, and the screen is always at hex 0400 unless you move it. There's a bank of memory at $C000 to $CFFF that is not used by the computer system; it's useful for staging machine language coding.

The Commodore 64 is the first Commodore machine in which it is sometimes desirable to write totally in machine language, with no BASIC at all. BASIC can be swapped out to release extra RAM, and large applications (word processors, spread sheets, and so on) are likely to do this.

Commodore PLUS/4

Similar to the Commodore 64 in many ways. The processor is a 7501, which has the same instruction set as the 6502. Screen memory and BASIC RAM have been moved a little higher. BASIC itself is greatly expanded.

Color and sound are implemented differently to the Commodore 64.

There's a built-in machine language monitor with expanded features, such as assemble and disassemble. This one is convenient for machine language programmers.

The memory arrangement is more sophisticated than on previous machines; large implementations may require insight into the machine's detailed architecture.

B Series

The B-128, B-256, CBM-128, and CBM-256 were designed as successors to the 80-column PET/CBM units. Architecture has been radically changed: the processor is a 6509, memory is bank switched, and zero page is significantly different from that of other models

The cassette buffer is no longer at $0330, so that the examples given in this book will need to be moved to a new part of RAM (addresses $0400 to $07FF are available). Bank switching is more complex than on other models. Beginners will find that there are more things to be kept

track of in this machine. If possible, beginners should try to find a simpler computer on which to take their first steps.

Implementation of large-scale programs require setting up a "transfer sequence" program to link the program's memory bank to that of the kernal. Usually, a bootstrap program will be needed to set everything up.

A machine language monitor is built into this line of machines. A few new commands have been made available: **. V** to switch banks, **. (α** to test disk status.

C

Memory Maps

A word about memory maps: they are always too big or too small for the use you have in mind.

The beginner may feel swamped by the wealth of detail. There's no threat, however. The information is there when you're ready for it. Browse through the information; it may be thought-provoking. Try reading or changing locations to see what happens.

The advanced programmer may want more: lengthy details on how each location is used, which parts of the system use these locations, and so on. Time and space don't permit such detail.

The maps are intended to be fairly complete. Those who want more detail may find them cryptic; but at least each location will be associated with a type of activity. Different machines may be compared by checking their respective maps. In some cases. programs may be converted with their use, since they will help to find the corresponding memory location in the target machine.

When you see a reference to a POKE or PEEK location—in this book or from other sources—check it in these maps. They will help add perspective.

"Original ROM" PET
The Great Zero-Page Hunt

Most users help themselves to the high part of the input buffer ($0040 to $0059, which is not used except when long lines of data are inputted.

Most zero-page locations may be copied to another part of memory so that their original contents can be restored after use. However, the programmer should take great care in modifying the following locations, which are critical within the operating system or BASIC. $03, $05, $64 to $67, $7A to $87, $89, $A2 to $A3, $B7, $C2 to $D9, $E0 to $E2, $F5.

Memory Map

Hex	Decimal	Description
0000–0002	0–2	USR jump
0003	3	Current I/O – prompt suppress
0005	5	Cursor control position
0008–0009	8–9	Integer value (for SYS, GOTO, and so on)
000A–0059	10–89	Input buffer
005A	90	Search character
005B	91	Scan-between-quotes flag
005C	92	Input buffer pointer; number of subscripts
005D	93	Default DIM flag
005E	94	Type: FF = string; 00 = numeric
005F	95	Type: 80 = integer; 00 = floating point
0060	96	Flag: DATA scan; LIST quote; memory
0061	97	Subscript flag; FNX flag
0062	98	0 = INPUT; $40 = GET; $98 = READ
0063	99	ATN sign/comparison evaluation flag
0064	100	Input flag (suppress output)
0065–0067	101–103	Pointers for descriptor stack

Hex	Decimal	Description
0068–0070	104–112	Descriptor stack (temporary strings)
0071–0074	113–116	Utility pointer area
0075–0078	117–120	Product area for multiplication
007A–007B	122–123	Pointer: start-of-BASIC
007C–007D	124–125	Pointer: start-of-variables
007E–007F	126–127	Pointer: start-of-arrays
0080–0081	128–129	Pointer: end-of-arrays
0082–0083	130–131	Pointer: string-storage (moving down)
0084–0085	132–133	Utility string pointer
0086–0087	134–135	Pointer: limit-of-memory
0088–0089	136–137	Current BASIC line number
008A–008B	138–139	Previous BASIC line number
008C–008D	140–141	Pointer:BASIC statement for CONT
008E–008F	142–143	Current DATA line number
0090–0091	144–145	Current DATA address
0092–0093	146–147	Input vector
0094–0095	148–149	Current variable name
0096–0097	150–151	Current variable address
0098–0099	152–153	Variable pointer for FOR/NEXT
009A–009B	154–155	Y-save; op-save; BASIC pointer save
009C	156	Comparison symbol accumulator
009D–00A2	157–162	Miscellaneous work area, pointers, and so on
00A3–00A5	163–165	Jump vector for functions
00A6–00AF	166–175	Miscellaneous numeric work area
00B0	176	Accum#1: exponent
00B1–00B4	177–180	Accum#1: mantissa
00B5	181	Accum#1:sign
00B6	182	Series evaluation constant pointer
00B7	183	Accum#1 hi-order (overflow)
00B8–00BD	184–189	Accum#2: exponent, and so on
00BE	190	Sign comparison, Acc#1 versus #2
00BF	191	Accum#1 lo-order (rounding)

Hex	Decimal	Description
00C0-00C1	192-193	Cassette buffer length/series pointer
00C2-00D9	194-217	CHRGET subroutine; get BASIC character
00C9-00CA	201-202	BASIC pointer (within subroutine)
00DA-00DE	218-222	Random number seed
00E0-00E1	224-225	Pointer to screen line
00E2	226	Position of cursor on above line
00E3-00E4	227-228	Utility pointer: tape, scroll
00E5-00E6	229-230	Tape end address/end of current program
00E7-00E8	231-232	Tape timing constants
00E9	233	Tape buffer character
00EA	234	Direct/programmed cursor: 0 = direct
00EB	235	Tape read timer 1 enabled
00EC	236	EOT received from tape
00ED	237	Read character error
00EE	238	Number of characters in file name
00EF	239	Current file logical address
00F0	240	Current file secondary address
00F1	241	Current file device number
00F2	242	Line margin
00F3-00F4	243-244	Pointer: start of tape buffer
00F5	245	Line where cursor lives
00F6	246	Last key/checksum/miscellaneous
00F7-00F8	247-248	Tape start address
00F9-00FA	249-250	File name pointer
00FB	251	Number of INSERTs outstanding
00FC	252	Write shift word/read character in
00FD	253	Tape blocks remaining to write/read
00FE	254	Serial word buffer
0100-010A	256-266	STR$ work area
0100-013E	256-318	Tape read error log
0100-01FF	256-511	Processor stack

Hex	Decimal	Description
0200–0202	512–513	Jiffy clock for TI and TI$
0203	515	Which key down: 255 = no key
0204	516	Shift key: 1 if depressed
0205–0206	517–518	Correction clock
0207–0208	519–520	Cassette status, #1 and #2
0209	521	Keyswitch PIA: STOP and RVS flags
020A	522	Timing constant for tape
020B	523	Load = 0; verify = 1
020C	524	Status word ST
020D	525	Number of characters in keyboard buffer
020E	526	Screen reverse flag
020F–0218	527–536	Keyboard input buffer
0219–021A	537–538	IRQ vector
021B–021C	539–540	BRK interrupt vector
021D	541	IEEE output: 255 = character pending
021E	542	End-of-line-for-input pointer
0220–0221	544–545	Cursor log (row, column)
0222	546	IEEE output buffer
0223	547	Key image
0224	548	0 = flash cursor
0225	549	Cursor timing countdown
0226	550	Character under cursor
0227	551	Cursor in blink phase
0228	552	EOT received from tape
0229–0241	553–577	Screen line wrap table
0242–024B	578–587	File logical address table
024C–0255	588–597	*File device number table*
0256–025F	598–607	File secondary address table
0260	608	Input from screen/from keyboard
0261	609	X save
0262	610	How many open files
0263	611	Input device, normally 0
0264	612	Output CMD device, normally 3
0265	613	Tape character parity
0266	614	Byte-received flag
0268–0269	615–616	File name pointer; counter

Hex	Decimal	Description
026C	620	Serial bit count
026F	623	Cycle counter
0270	624	Tape writer countdown
0271–0272	625–626	Tape buffer pointers, #1 and #2
0273	627	Write leader count; read pass 1/2
0274	628	Write new byte; read error flag
0275	629	Write start bit; read bit sequence error
0276–0277	630–631	Error log pointers, pass 1/2
0278	632	0 = scan/1–15 = count/ $40 = load/$80 = end
0279	633	Write leader length; read checksum
027A–0339	634–825	Tape#1 input buffer
033A–03F9	826–1017	Tape#2 input buffer
03FA–03FB	1018–1019	Monitor extension vector
0400–7FFF	1024–32767	Available RAM including expansion
8000–83E7	32768–33767	Screen RAM memory
C000–E7F8	49152–59384	BASIC ROM; part of kernal ROM
E810–E813	59408–59411	PIA 1 (6520)-keyboard interface
E820–E823	59424–59427	PIA 2 (6520)-IEEE interface
E840–E84F	59456–59471	VIA (6522)-Miscellaneous interface, timers
F000–FFFF	61440–65535	Kernal ROM routines.

PIA and VIA charts are the same as shown for Upgrade/4.0 units.

UPGRADE and BASIC 4.0 Systems

The Great Zero-Page Hunt

Zero-page locations are tough to find in these areas. Locations $1F to $27, $4B to $50, and $54 to $5D are work areas available for temporary use. If tape is not being read or written, addresses $B1 to $C3 are available.

Most zero-page locations may be copied to another part of memory so that their original contents can be restored after use. The programmer should take great care, however, in modifying the following locations, which are critical within the operating system of BASIC: $10, $13 to $15, $28 to $35, $37, $50 to $51, $65, $70 to $87, $8D to $B0, $C4 to $FA.

Memory Map

Where Upgrade ROM differs from 4.0, an asterisk (*) is shown and the 4.0 value is given. There are some differences in usage between the 40- and 80-column machines.

Hex	Decimal	Description
0000–0002	0–2	USR jump
0003	3	Search character
0004	4	Scan-between-quotes flag
0005	5	Input buffer pointer; number of subscripts
0006	6	Default DIM flag
0007	7	Type: FF = string; 00 = numeric
0008	8	Type: 80 = integer; 00 = floating point
0009	9	Flag: DATA scan; LIST quote; memory
000A	10	Subscript flag; FNX flag
000B	11	0 = INPUT; $40 = GET; $98 = READ
000C	12	ATN sign/comparison evaluation flag
000D–000F	13–15	*Disk status DS$ descriptor
0010	16	*Current I/O device for prompt-suppress
0011–0012	17–18	Integer value (for SYS, GOTO, and so on)
0013–0015	19–21	Pointers for descriptor stack
0016–001E	22–30	Descriptor stack (temporary strings)
001F–0022	31–34	Utility pointer area
0023–0027	35–39	Product area for multiplication
0028–0029	40–41	Pointer: start-of-BASIC
002A–002B	42–43	Pointer: start-of-variables
002C–002D	44–45	Pointer: start-of-arrays

Hex	Decimal	Description
002E–002F	46–47	Pointer: end-of-arrays
0030–0031	48–49	Pointer: string-storage (moving down)
0032–0033	50–51	Utility string pointer
0034–0035	52–53	Pointer: limit-of-memory
0036–0037	54–55	Current BASIC line number
0038–0039	56–57	Previous BASIC line number
003A–003B	58–59	Pointer: BASIC statement for CONT
003C–003D	60–61	Current DATA line number
003E–003F	62–63	Current DATA address
0040–0041	64–65	Input vector
0042–0043	66–67	Current variable name
0046–0047	70–71	Variable pointer for FOR/NEXT
0048–0049	72–73	Y-save; op-save; BASIC pointer save
004A	74	Comparison symbol accumulator
004B–0050	75–80	Miscellaneous work area, pointers, and so on
0051–0053	81–83	Jump vector for functions
0054–005D	84–93	Miscellaneous numeric work area
005E	94	Accum#1: exponent
005F–0062	95–98	Accum#1: mantissa
0063	99	Accum#1: sign
0064	100	Series evaluation constant pointer
0065	101	Accum#1 hi-order (overflow)
0066–006B	102–107	Accum#2: exponent, and so on
006C	108	Sign comparison, Acc#1 versus #2
006D	106	Accum#1 lo-order (rounding)
006E–006F	110–111	Cassette buffer length/series pointer
0070–0087	112–135	CHRGET subroutine; get BASIC character
0077–0078	119–120	BASIC pointer (within subroutine)
0088–008C	136–140	Random number seed
008D–008F	141–143	Jiffy clock for TI and TI$

Hex	Decimal	Description
0090-0091	144-145	IRQ vector
0092-0093	146-147	BRK interrupt vector
0094-0095	148-149	NMI interrupt vector
0096	150	Status word ST
0097	151	Which key down: 255 = no key
0098	152	Shift key: 1 if depressed
0099-009A	153-154	Correction clock
009B	155	Keyswitch PIA: STOP and RVS flags
009C	156	Timing constant for tape
009D	157	Load = 0; verify = 1
009E	158	Number of characters in keyboard buffer
009F	159	Screen reverse flag
00A0	160	IEEE output: 255 = character pending
00A1	161	End-of-line-for-input pointer
00A3-00A4	163-164	Cursor log (row, column)
00A5	165	IEEE output buffer
00A6	166	Key image
00A7	167	0 = flash cursor
00A8	168	Cursor timing countdown
00A9	169	Character under cursor
00AA	170	Cursor in blink phase
00AB	171	EOT received from tape
00AC	172	Input from screen/from keyboard
00AD	173	X save
00AE	174	How many open files
00AF	175	Input device, normally 0
00B0	176	Output CMD device, normally 3
00B1	177	Tape character parity
00B2	178	Byte-received flag
00B3	179	Logical address temporary save
00B4	180	Tape buffer character; MLM command
00B5	181	File name pointer; MLM flag; counter
00B7	183	Serial bit count
00B9	185	Cycle counter
00BA	186	Tape writer countdown

Hex	Decimal	Description
00BB–00BC	187–188	Tape buffer pointers, #1 and #2
00BD	189	Write leader count; read pass 1/2
00BE	190	Write new byte; read error flag
00BF	191	Write start bit; read bit sequence error
00C0–00C1	192–193	Error log pointers, pass 1/2
00C2	194	0 = scan/1 – 15 = count/ $40 = load/$80 = end
00C3	195	Write leader length; read checksum
00C4–00C5	196–197	Pointer to screen line
00C6	198	Position of cursor on above line
00C7–00C8	199–200	Utility pointer: tape, scroll
00C9–00CA	201–202	Tape end address/end of current program
00CB–00CC	203–204	Tape timing constants
00CD	205	0 = direct cursor; else programmed
00CE	206	Tape read timer 1 enabled
00CF	207	EOT received from tape
00D0	208	Read character error
00D1	209	Number of characters in file name
00D2	210	Current file logical address
00D3	211	Current file secondary address
00D4	212	Current file device number
00D5	213	Right-hand window or line margin
00D6–00D7	214–215	Pointer: start of tape buffer
00D8	216	Line where cursor lives
00D9	217	Last key/checksum/ miscellaneous
00DA–00DB	218–219	File name pointer
00DC	220	Number of INSERTs outstanding
00DD	221	Write shift word/read character in
00DE	222	Tape blocks remaining to write/ read

Hex	Decimal	Description
OODF	223	Serial word buffer
OOEO-OOF8	224-248	(40-column) Screen line wrap table
OOEO-OOE1	224-225	*(80-column) Top, bottom of window
OOE2	226	*(80-column) Left window margin
OOE3	227	*(80-column) Limit of keyboard buffer
OOE4	228	*(80-column) Key repeat flag
OOE5	229	*(80-column) Repeat countdown
OOE6	230	*(80-column) New key marker
OOE7	231	*(80-column) Chime time
OOE8	232	*(80-column) HOME count
OOE9-OOEA	233-234	*(80-column) Input vector
OOEB-OOEC	235-236	*(80-column) Output vector
OOF9-OOFA	249-250	Cassette status, #1 and #2
OOFB-OOFC	251-252	MLM pointer/tape start address
OOFD-OOFE	253-254	MLM, DOS pointer, miscellaneous
0100-010A	256-266	STR$ work area, MLM work
0100-013E	256-318	Tape read error log
0100-01FF	256-511	Processor stack
0200-0250	512-592	MLM work area; input buffer
0251-025A	593-602	File logical address table
025B-0264	603-612	File device number table
0265-026E	613-622	File secondary address table
026F-0278	623-632	Keyboard input buffer
027A-0339	634-825	Tape#1 input buffer
033A-03F9	826-1017	Tape#2 input buffer
033A-0380	826-896	*DOS work area
03E9	1001	(Fat 40) New key marker
03EA	1002	(Fat 40) Key repeat countdown
03EB	1003	(Fat 40) Keyboard buffer limit
03EC	1004	(Fat 40) Chime time
03ED	1005	(Fat 40) Decisecond timer
03EE	1006	(Fat 40) Key repeat flag
03EE-03F7	1006-1015	(80-column) Tab stop table
03EF	1007	(Fat 40) Tab work value
03F0-9	1008-1017	(Fat 40) Tab stops
03FA-03FB	1018-1019	Monitor extension vector

Hex	Decimal	Description
03FC	1020	*IEEE timeout defeat
0400-7FFF	1024-32767	Available RAM including expansion
8000-83E7	32768-33767	(40-column) Video RAM
8000-87CF	32768-34767	*(80-column) Video RAM
9000-aFFF	36864-45055	Available ROM expansion area
B000-E7FF	45056-59391	BASIC ROM, part of kernal
E810-E813	59408-59411	PIA 1-keyboard I/O
E820-E823	59424-59427	PIA 2-IEEE-488 I/O
E840-E84F	59456-59471	VIA-I/O and timers
E880-E881	59520-59521	(80-column and Fat 40) CRT controller
F000-FFFF	61440-65535	Kernal ROM

					6520	
E810	Diag Sens/ Uncrash	EOI in	Tape Switch Sense #1 #2		Keyboard Row Select	59408
E811	Tape#1 In Latch		(Screen Blank—Orig ROM) EOI Out	DDRA Access	Tape#1 Input L Control	59409
E812			Keyboard Input for selected row			59410
E813	Retrace Latch		Cassette#1 Motor Output	DDRB Access	Retrace Interrupt Control	59411

Figure C.1
PIA 1 chart

					6520	
E820			IEEE-488 Input			59424
E821	ATN Int		NDAC Out	DDRA Access	ATN Int Control	59425
E822			IEEE-488 Output			59426
E823	SRQ Int		DAV Out	DDRB Access	SQR Int Control	59427

Figure C.2
PIA 2 chart

						6522			
E840	$\overline{DAV}$ In	$\overline{NRFD}$ In	Retrace In	Tape #2 Motor	Tape Output	$\overline{ATN}$ Out	NRFD Out	$\overline{NDAC}$ In	59456
E841	Unused (See E84F)								59457
E842	Data Direction Register B (for E840)								59458
E843	Data Direction Register A (for E84F)								59459
E844	Timer 1								59460
E845									59461
E846	Timer 1 Latch								59462
E847									59463
E848	Timer 2								59464
E849									59465
E84A	Shift Register (unused)								59466
E84B	T1 Control		T2 Cont	Shift Register Control			Latch Controls PB PA		59467
E84C	CB2 (PUP) Control		CB1 Cntl Tape#2	CA2 Control Graphics/Text Mode			CA1 (PUP) Control		59468
E84D	Irq Stats ---- Int Enabl	Timer 1 Int	Timer 2 Int	CB1 Tape#2 Int	CB2 (PUB) Int	SR Unused	CA1 (PUP) Int	CA2 G/T Mode unused	59469
E84E									59470
E846F	Parallel User Port Data Register PA								59471

Figure C.3
VIA chart

CBM 8032 and FAT-40 6545 CRT Controller

NOTES: 1. Registers are write-only.
2. Avoid extreme changes in Register 0. CRT damage could result.
3. Register 0 will adjust scan to allow interfacing to external monitor.
4. Register 12, Bit 4, will "invert" the video signal.
5. Register 12, Bit 5, switches to an alternate character set. The character set is not implemented on most machines except Super-PET.

TYPICAL VALUES
(DECIMAL)

$E840 $E841   TEXT   GRAPHICS

		TEXT	GRAPHICS
0	HORIZONTAL TOTAL	49	49
1	HOR. CHAR. DISPLAYED	40	40
2	H. SYNC POSITION	41	41
3	V SYNC WIDTH H	15	15
4	VERTICAL TOTAL	32	40
5	VERT. TOT. ADJUST	3	5
6	VERTICAL DISPLAYED	25	25
7	VERT. SYNC POSITION	29	33
8	MODE	0	0
9	SCAN LINES	9	7
10	CURSOR START (UNUSED)	0	0
11		0	0
12	C R DISPLAY	16	16
13	ADDRESS	0	0

NOTES: 1. REGISTERS ARE WRITE-ONLY
        2. AVOID EXTREME CHANGES IN
           REGISTER, OR CRT DAMAGE
           COULD RESULT
        3. REGISTER 0 WILL ADJUST SCAN
           TO ALLOW INTERFACING TO
           EXTERNAL MONITOR
        4. REGISTER 12, BIT 4, WILL "INVERT"
           THE VIDEO SIGNAL.
        5. REGISTER 12, BIT 5, SWITCHES TO
           AN ALTERNATE CHARACTER SET.
           THE CHARACTER SET IS NOT
           IMPLEMENTED ON MOST MACHINES
           EXCEPT SUPER-PET.

Figure C.4

The 6522 VIA

6522

E840	$\overline{\text{DAV In}}$	$\overline{\text{NRFD In}}$	Retrace In	Tape #2 Motor	Tape Output	$\overline{\text{ATN Out}}$	NRFD Out	$\overline{\text{NDAC In}}$	59456
E841	Unused (See E84F)								59457
E842	Data Direction Register B (for E840)								59458
E843	Data Direction Register A (for E84F)								59459
E844	Timer 1								59460
E845									59461
E846	Timer 1 Latch								59462
E847									59463
E848	Timer 2								59464
E849									59465
E84A	Shift Register (unused)								59466
E84B	T1 Control		T2 Cont	Shift Register Control			Latch Controls PB PA		59467
E84C	CB2 (PUP) Control			CB1 Cntl Tape#2	CA2 Control Graphics/Text Mode			CA1 (PUP) Control	59468
E84D E84E	Irq Stats ---- Int Enabl	Timer 1 Int	Timer 2 Int	CB1 Tape#2 Int	CB2 (PUB) Int	SR Unused	CA1 (PUP) Int	CA2 G/T Mode unused	59469 59470
E846F	Parallel User Port Data Register PA								59471

Figure C.5
VIA chart

VIC-20
The Great Zero-Page Hunt

Locations $FC to $FF are available. Locations $22 to $2A, $4E to $53, and $57 to $60 are work areas available for temporary use.

Most zero-page locations may be copied to another part of memory so that their original contents can be restored after use. However, the programmer should take great care in modifying the following locations, which are critical within the operating system or BASIC: $13, $16 to $18, $2B to $38, $3A, $53 to $54, $68, $73 to $8A, $90 to $9A, $A0 to $A2, $B8 to $BA, $C5 to $F4.

Memory Map

Hex	Decimal	Description
0000-0002	0-2	USR jump
0003-0004	3-4	Float-fixed vector
0005-0006	5-6	Fixed-float vector
0007	7	Search character
0008	8	Scan-quotes flag
0009	9	TAB column save
000A	10	0=LOAD, 1=VERIFY
000B	11	Input buffer pointer/number of subscripts
000C	12	Default DIM flag
000D	13	Type: FF=string; 00=numeric
000E	14	Type: 80=integer; 00=floating point
000F	15	DATA scan/LIST quote/memory flag
0010	16	Subscript/FNx flag
0011	17	0=INPUT; $40=GET; $98=READ
0012	18	ATN sign/Comparison evaluation flag
0013	19	Current I/O prompt flag
0014-0015	20-21	Integer value
0016	22	Pointer: temporary string stack
0017-0018	23-24	Last temporary string vector
0019-0021	25-33	Stack for temporary strings
0022-0025	34-37	Utility pointer area
0026-002A	38-42	Product area for multiplication
002B-002C	43-44	Pointer: start-of-BASIC
002D-002E	45-46	Pointer: start-of-variables
002F-0030	47-48	Pointer: start-of-arrays
0031-0032	49-50	Pointer: end-of-arrays
0033-0034	51-52	Pointer: string-storage (moving down)
0035-0036	53-54	Utility string pointer
0037-0038	55-56	Pointer: limit-of-memory
0039-003A	57-58	Current BASIC line number
003B-003C	59-60	Previous BASIC line number
003D-003E	61-62	Pointer: BASIC statement for CONT

Hex	Decimal	Description
0097	151	Register save
0098	152	How many open files
0099	153	Input device, normally 0
009A	154	Output CMD device, normally 3
009B	155	Tape character parity
009C	156	Byte-received flag
009D	157	Direct = $80/RUN = 0 output control
009E	158	Tape pass 1 error log/character buffer
009F	159	Tape pass 2 error log corrected
00A0–00A2	160–162	Jiffy Clock HML
00A3	163	Serial bit count/EOI flag
00A4	164	Cycle count
00A5	165	Countdown, tape write/bit count
00A6	166	Tape buffer pointer
00A7	167	Tape write leader count/read pass/inbit
00A8	168	Tape write new byte/read error/inbit count
00A9	169	Write start bit/read bit error/stbit
00AA	170	Tape Scan;Cnt;Load;End/byte assembly
00AB	171	Write lead length/read checksum/parity
00AC–00AD	172–173	Pointer: tape buffer, scrolling
00AE–00AF	174–175	Tape end address/end of program
00B0–00B1	176–177	Tape timing constants
00B2–00B3	178–179	Pointer: start of tape buffer
00B4	180	1 = tape timer enabled; bit count
00B5	181	Tape EOT/RS232 next bit to send
00B6	182	Read character error/outbyte buffer
00B7	183	Number of characters in file name
00B8	184	Current logical file
00B9	185	Current secondary address
00BA	186	Current device
00BB–00BC	187–188	Pointer to file name

Hex	Decimal	Description
003F-0040	63-64	Current DATA line number
0041-0042	65-66	Current DATA address
0043-0044	67-68	Input vector
0045-0046	69-70	Current variable name
0047-0048	71-72	Current variable address
0049-004A	73-74	Variable pointer for FOR/NEXT
004B-004C	75-76	Y-save; op-save; BASIC pointer save
004D	77	Comparison symbol accumulator
004E-0053	78-83	Miscellaneous work area, pointers, and so on
0054-0056	84-86	Jump vector for functions
0057-0060	87-96	Miscellaneous numeric work area
0061	97	Accum#1· exponent
0062-0065	98-101	Accum#1: mantissa
0066	102	Accum#1: sign
0067	103	Series evaluation constant pointer
0068	104	Accum#1 hi-order (overflow)
0069-006E	105-110	Accum#2: exponent, and so on
006F	111	Sign comparison, Acc#1 versus #2
0070	112	Accum#1 lo-order (rounding)
0071-0072	113-114	Cassette buffer length/series pointer
0073-008A	115-138	CHRGET subroutine; get BASIC character
007A-007B	122-123	BASIC pointer (within subroutine)
008B-008F	139-143	RND seed value
0090	144	Status word ST
0091	145	Keyswitch PIA: STOP and RVS flags
0092	146	Timing constant for tape
0093	147	Load = 0; verify = 1
0094	148	Serial output: deferred character flag
0095	149	Serial deferred character
0096	150	Tape EOT received

Hex	Decimal	Description
00BD	189	Write shift word/`read input char`
00BE	190	Number of blocks remaining to write/read
00BF	191	Serial word buffer
00C0	192	Tape motor interlock
00C1–00C2	193–194	I/O start address
00C3–00C4	195–196	Kernal setup pointer
00C5	197	Last key pressed
00C6	198	Number of characters in keyboard buffer
00C7	199	Screen reverse flag
00C8	200	End-of-line for input pointer
00C9–00CA	201–202	Input cursor log (row, column)
00CB	203	Which key: 64 if no key
00CC	204	0 = flash cursor
00CD	205	Cursor timing countdown
00CE	206	Character under cursor
00CF	207	Cursor in blink phase
00D0	208	Input from screen/from keyboard
00D1–00D2	209–210	Pointer to screen line
00D3	211	Position of cursor on above line
00D4	212	0 = direct cursor; else programmed
00D5	213	Current screen line length
00D6	214	Row where cursor lives
00D7	215	Last inkey/`checksum`/buffer
00D8	216	Number of `INSERT`s outstanding
00D9–00F0	217–240	Screen link table
00F1	241	Dummy screen link
00F2	242	Screen row marker
00F3–00F4	243–244	Screen color pointer
00F5–00F6	245–246	Keyboard pointer
00F7–00F8	247–248	RS-232 Rcv pntr
00F9–00FA	249–250	RS-232 Tx pntr
00FF–010A	256–266	Floating to ASCII work area
0100–103E	256–318	Tape error log
0100–01FF	256–511	Processor stack area
0200–0258	512–600	BASIC input buffer

Hex	Decimal	Description
0259–0262	601–610	Logical file table
0263–026C	611–620	Device number table
026D–0276	621–630	Secondary address table
0277–0280	631–640	Keyboard buffer
0281–0282	641–642	Start of BASIC memory
0283–0284	643–644	Top of BASIC memory
0285	645	Serial bus timeout flag
0286	646	Current color code
0287	647	Color under cursor
0288	648	Screen memory page
0289	649	Maximum size of keyboard buffer
028A	650	Repeat all keys
028B	651	Repeat speed counter
028C	652	Repeat delay counter
028D	653	Keyboard shift/control flag
028E	654	Last shift pattern
028F–0290	655–656	Keyboard table setup pointer
0291	657	Keymode (Kattacanna)
0292	658	0 = scroll enable
0293	659	RS-232 chip control
0294	660	RS-232 chip command
0295–0296	661–662	Bit timing
0297	663	RS-232 status
0298	664	Number of bits to send
0299–029A	665	RS-232 speed/code
029B	667	RS-232 receive pointer
029C	668	RS-232 input pointer
029D	669	RS-232 transmit pointer
029E	670	RS-232 output pointer
029F–02A0	671–672	IRQ save during tape I/O
0300–0301	768–769	Error message link
0302–0303	770–771	BASIC warm start link
0304–0305	772–773	Crunch BASIC tokens link
0306–0307	774–775	Print tokens link
0308–0309	776–777	Start new BASIC code link
030A–030B	778–779	Get arithmetic element link
030C	780	SYS A-reg save
030D	781	SYS X-reg save
030E	782	SYS Y-reg save
030F	783	SYS status reg save

Hex	Decimal	Description
0314-0315	788-789	IRQ vector (EABF)
0316-0317	790-791	Break interrupt vector (FED2)
0318-0319	792-793	NMI interrupt vector (FEAD)
031A-031B	794-795	OPEN vector (F40A)
031C-031D	796-797	CLOSE vector (F34A)
031E-031F	798-799	Set-input vector (F2C7)
0320-0321	800-801	Set-output vector (F309)
0322-0323	802-803	Restore I/O vector (F3F3)
0324-0325	804-805	INPUT vector (F20E)
0326-0327	806-807	Output vector (F27A)
0328-0329	808-809	Test-STOP vector (F770)
032A-032B	810-811	GET vector (F1F5)
032C-032D	812-813	Abort I/O vector (F3EF)
032E-032F	814-815	USR vector (FED2)
0330-0331	816-817	LOAD link
0332-0333	818-819	SAVE link
033C-03FB	828-1019	Cassette buffer
0400-0FFF	1024-4095	3K RAM expansion area
1000-1FFF	4096-8191	Normal BASIC memory
2000-7FFF	8192-32767	Memory expansion area
8000-8FFF	32768-36863	Character bit maps (ROM)
9000-900F	36864-36879	Video interface chip (6560)
9110-912F	37136-37151	VIA (6522) interface-NMI
9120-912F	37152-37167	VIA (6522) interface-IRQ
9400-95FF	37888-38399	Alternate color nybble area
9600-97FF	38400-38911	Main color nybble area
A000-BFFF	40960-49151	Plug-in ROM area
C000-FFFF	49152-65535	ROM: BASIC and operating system

VIC 6560 Chip

Address			
$9000	Inter-lace	Left Margin (= 5)	36864
$9001		Top Margin (= 25)	36865
$9002	Scrn Ad bit 9	# Columns (= 22)	36866
$9003	bit 0	# Rows (= 23) · Double Char	36867
$9004		Input Raster Value: bits 8–1	36868
$9005	Screen Address bits 13–10	Character Address bits 13–10	36869
$9006	Light Pen Input	Horizontal	36870
$9007		Vertical	36871
$9008	Paddle Inputs	X	36872
$9009		Y	36873
$900A	ON	Voice 1	36874
$900B	ON	Voice 2 Frequency	36875
$900C	ON	Voice 3	36876
$900D	ON	Noise	36877
$900E	Multi-Colour Mode (= 0)	Sound Amplitude	36878
$900F	Screen Background Color	Foregnd /Backg · Frame Color	36879

Figure C.6

VIC 6522 Usage

	DSR in	CTS in		DCD* in	RI* in	DTR out	RTS out	Data in	
$9110	RS-232 Interface or, Parallel User Port								37136
$9111	Unused — see $911F								37137
$9112	DDRB (for $9110)								37138
$9113	DDRA (for $911F)								37139
$9114	T1-L	RS-232 Send Speed; Tape Write Timing						—	37140
$9115	T1-H							—	37141
$9116	T1 Latch L								37142
$9117	T1 Latch H								37143
$9118	T2-L	RS-232 Input timing							37144
$9119	T2-H								37145
$911A	Shift Register (*unused)								37146
$911B	T1 Control	T2 Cnt	Shift Reg Control			PB LE	PA LE		37147
$911C	CB2 RS-232 Send		CB1 C	CA2 Tape motor ctrl			CA1 Ctl		37148
$911D	NMI·	T1	T2	CB1: RS-232 in		CA1. Restore button			37149
$911E									37150
$911F	ATN out	Tape sense	Button	--------- Joysticks --------- Left	Down	Up	Serial Data in	Serial Clk in	37151

Figure C.7

VIC 6522 Usage

Addr						Dec
$9120	Joystk Right		Tape Out			37152
	Keyboard Row Select					
$9121	Keyboard Column Input					37153
$9122	DDRB (for $9120)					37154
$9123	DDRA (for $9121)					37155
$9124	T1-L	Cassette Tape Read;				37156
$9125	T1-H	Keyboard & Clock				37157
$9126	T1-L Latch	Interrupt Timing				37158
$9127	T1-H Latch					37159
$9128	T2-L	Serial Bus Timing				37160
$9129	T2-H	Tape R/W Timing				37161
$912A	Shift Register (*Unused)					37162
$912B	T1 Control	T2 Ctrl	Shift Register Contrl	PB LE	PA LE	37163
$912C	Serial Bus Data Out	CB1 Contl	Serial Clock Line out	CA1 Contl		37164
$912D			CB1:*		CA1:	37165
$912E	IRQ.	T1	T2	SRQ in	Tape in	37166
$912F	*Unused: see $9121					37167

Figure C.8

Commodore 64:

The Great Zero-Page Hunt:

Locations $FC to $FF are available. Locations $22 to $2A, $4E to $53, and $57 to $60 are work areas available for temporary use.

Most zero-page locations may be copied to another part of memory so that their original contents can be restored after use. The programmer should take great care, however, in modifying the following locations, which are critical within the operating system or BASIC: $13, $16 to $18, $2B to $38, $3A, $53 to $54, $68, $73 to $8A, $90 to $9A, $A0 to $A2, $B8 to $BA, $C5 to $F4.

Memory Map

Hex	Decimal	Description
0000	0	Chip directional register
0001	1	Chip I/O; memory and tape control
0003–0004	3–4	Float-fixed vector
0005–0006	5–6	Fixed-float vector
0007	7	Search character
0008	8	Scan-quotes flag
0009	9	TAB column save
000A	10	0 = LOAD, 1 = VERIFY
000B	11	Input buffer pointer/number of subscripts
000C	12	Default DIM flag
000D	13	Type: FF = string; 00 = numeric
000E	14	Type: 80 = integer; 00 = floating point
000F	15	DATA scan/LIST quote/memory flag
0010	16	Subscript/FNx flag
0011	17	0 = INPUT;$40 = GET; $98 = READ
0012	18	ATN sign/Comparison evaluation flag
0013	19	Current I/O prompt flag
0014–0015	20–21	Integer value

Hex	Decimal	Description
0016	22	Pointer· temporary string stack
0017-0018	23-24	Last temporary string vector
0019-0021	25-33	Stack for temporary strings
0022-0025	34-37	Utility pointer area
0026-002A	38-42	Product area for multiplication
002B-002C	43-44	Pointer: start-of-BASIC
002D-002E	45-46	Pointer: start-of-variables
002F-0030	47-48	Pointer: start-of-arrays
0031-0032	49-50	Pointer: end-of-arrays
0033-0034	51-52	Pointer: string-storage (moving down)
0035-0036	53-54	Utility string pointer
0037-0038	55-56	Pointer: limit-of-memory
0039-003A	57-58	Current BASIC line number
003B-003C	59-60	Previous BASIC line number
003D-003E	61-62	Pointer: BASIC statement for CONT
003F-0040	63-64	Current DATA line number
0041-0042	65-66	Current DATA address
0043-0044	67-68	Input vector
0045-0046	69-70	Current variable name
0047-0048	71-72	Current variable address
0049-004A	73-74	Variable pointer for FOR/NEXT
004B-004C	75-76	Y-save; op-save; BASIC pointer save
004D	77	Comparison symbol accumulator
004E-0053	78-83	Miscellaneous work area, pointers, and so on
0054-0056	84-86	Jump vector for functions
0057-0060	87-96	Miscellaneous numeric work area
0061	97	Accum#1: exponent
0062-0065	98-101	Accum#1: mantissa
0066	102	Accum#1: sign
0067	103	Series evaluation constant pointer
0068	104	Accum#1 hi-order (overflow)
0069-006E	105-110	Accum#2: exponent, and so on
006F	111	Sign comparison, Acc#1 versus #2
0070	112	Accum#1 lo-order (rounding)

Hex	Decimal	Description
0071-0072	113-114	Cassette buffer length/series pointer
0073-008A	115-138	CHRGET subroutine; get BASIC character
007A-007B	122-123	BASIC pointer (within subroutine)
008B-008F	139-143	RND seed value
0090	144	Status word ST
0091	145	Keyswitch PIA: STOP and RVS flags
0092	146	Timing constant for tape
0093	147	Load = 0; verify = 1
0094	148	Serial output: deferred character flag
0095	149	Serial deferred character
0096	150	Tape EOT received
0097	151	Register save
0098	152	How many open files
0099	153	Input device, normally 0
009A	154	Output CMD device, normally 3
009B	155	Tape character parity
009C	156	Byte-received flag
009D	157	Direct = $80/RUN = 0 output control
009E	158	Tape pass 1 error log/character buffer
009F	159	Tape pass 2 error log corrected
00A0-00A2	160-162	Jiffy Clock HML
00A3	163	Serial bit count/EOI flag
00A4	164	Cycle count
00A5	165	Countdown, tape write/bit count
00A6	166	Tape buffer pointer
00A7	167	Tape write leader count/read pass/inbit
00A8	168	Tape write new byte/read error/inbit count
00A9	169	Write start bit/read bit error/stbit
00AA	170	Tape Scan;Cnt;Load; End/byte assembly
00AB	171	Write lead length/read checksum/parity
00AC-00AD	172-173	Pointer: tape buffer, scrolling

Hex	Decimal	Description
00AE–00AF	174–175	Tape end address/end of program
00B0–00B1	176–177	Tape timing constants
00B2–00B3	178–179	Pointer: start of tape buffer
00B4	180	1 = tape timer enabled; bit count
00B5	181	Tape EOT/RS232 next bit to send
00B6	182	Read character error/outbyte buffer
00B7	183	Number of characters in file name
00B8	184	Current logical file
00B9	185	Current secondary address
00BA	186	Current device
00BB–00BC	187–188	Pointer to file name
00BD	189	Write shift word/read input char
00BE	190	Number of blocks remaining to write/read
00BF	191	Serial word buffer
00C0	192	Tape motor interlock
00C1–00C2	193–194	I/O start address
00C3–00C4	195–196	Kernel setup pointer
00C5	197	Last key pressed
00C6	198	Number of characters in keyboard buffer
00C7	199	Screen reverse flag
00C8	200	End-of-line for input pointer
00C9–00CA	201–202	Input cursor log (row, column)
00CB	203	Which key: 64 if no key
00CC	204	0 = flash cursor
00CD	205	Cursor timing countdown
00CE	206	Character under cursor
00CF	207	Cursor in blink phase
00D0	208	Input from screen/from keyboard
00D1–00D2	209–210	Pointer to screen line
00D3	211	Position of cursor on above line
00D4	212	0 = direct cursor; else programmed
00D5	213	Current screen line length

Hex	Decimal	Description
`00D6`	`214`	Row where cursor lives
`00D7`	`215`	Last inkey/checksum/buffer
`00D8`	`216`	Number of `INSERT`s outstanding
`00D9-00F2`	`217-242`	Screen line link table
`00F3-00F4`	`243-244`	Screen color pointer
`00F5-00F6`	`245-246`	Keyboard pointer
`00F7-00F8`	`247-248`	RS-232 Rcv pntr
`00F9-00FA`	`249-250`	RS-232 Tx pntr
`00FF-010A`	`256-266`	Floating to ASCII work area
`0100-103E`	`256-318`	Tape error log
`0100-01FF`	`256-511`	Processor stack area
`0200-0258`	`512-600`	BASIC input buffer
`0259-0262`	`601-610`	Logical file table
`0263-026C`	`611-620`	Device number table
`026D-0276`	`621-630`	Secondary address table
`0277-0280`	`631-640`	Keyboard buffer
`0281-0282`	`641-642`	Start of BASIC memory
`0283-0284`	`643-644`	Top of BASIC memory
`0285`	`645`	Serial bus timeout flag
`0286`	`646`	Current color code
`0287`	`647`	Color under cursor
`0288`	`648`	Screen memory page
`0289`	`649`	Maximum size of keyboard buffer
`028A`	`650`	Repeat all keys
`028B`	`651`	Repeat speed counter
`028C`	`652`	Repeat delay counter
`028D`	`653`	Keyboard Shift/Control flag
`028E`	`654`	Last shift pattern
`028F-0290`	`655-656`	Keyboard table setup pointer
`0291`	`657`	Keyboard shift mode
`0292`	`658`	0 = scroll enable
`0293`	`659`	RS-232 control reg
`0294`	`660`	RS-232 command reg
`0295-0296`	`661-662`	Bit timing
`0297`	`663`	RS-232 status
`0298`	`664`	Number of bits to send
`0299-029A`	`665`	RS-232 speed/code
`029B`	`667`	RS232 receive pointer
`029C`	`668`	RS232 input pointer

Hex	Decimal	Description
029D	669	RS232 transmit pointer
029E	670	RS232 output pointer
029F-02A0	671-672	IRQ save during tape I/O
02A1	673	CIA 2 (NMI) interrupt control
02A2	674	CIA 1 timer A control log
02A3	675	CIA 1 interrupt Log
02A4	676	CIA 1 timer A enabled flag
02A5	677	Screen row marker
02C0-02FE	704-766	(Sprite 7)
0300-0301	768-769	Error message link
0302-0303	770-771	BASIC warm start link
0304-0305	772-773	Crunch BASIC tokens link
0306-0307	774-775	Print tokens link
0308-0309	776-777	Start new BASIC code link
030A-030B	778-779	Get arithmetic element link
030C	780	SYS A-reg save
030D	781	SYS X-reg save
030E	782	SYS Y-reg save
030F	783	SYS status reg save
0310-0312	784-785	USR function jump (B248)
0314-0315	788-789	IRQ vector (EA31)
0316-0317	790-791	Break interrupt vector (FE66)
0318-0319	792-793	NMI interrupt vector (FE47)
031A-031B	794-795	OPEN vector (F34A)
031C-031D	796-797	CLOSE vector (F291)
031E-031F	798-799	Set-input vector (F20E)
0320-0321	800-801	Set-output vector (F250)
0322-0323	802-803	Restore I/O vector (F333)
0324-0325	804-805	Input vector (F157)
0326-0327	806-807	Output vector (F1CA)
0328-0329	808-809	Test-STOP vector (F6ED)
032A-032B	810-811	GET vector (F13E)
032C-032D	812-813	Abort I/O vector (F32F)
032E-032F	814-815	USR vector (FE66)
0330-0331	816-817	LOAD link (F4A5)
0332-0333	818-819	SAVE link (F5ED)
033C-03FB	828-1019	Cassette buffer
0340-037E	832-894	(Sprite 13)
0380-03BE	896-958	(Sprite 14)
03C0-03FE	960-1022	(Sprite 15)
0400-07FF	1024-2047	Screen memory
0800-9FFF	2048-40959	BASIC ROM memory

Hex	Decimal	Description
8000-9FFF	32768-40959	Alternative: ROM plug-in area
A000-BFFF	40960-49151	ROM: BASIC
A000-BFFF	49060-59151	Alternate: RAM
C000-CFFF	49152-53247	RAM memory, including alternative
D000-D02E	53248-53294	Video chip (6566)
D400-D41C	54272-54300	Sound chip (6581 SID)
D800-DBFF	55296-56319	Color nybble memory
DC00-DC0F	56320-56335	Interface chip 1, IRQ (6526 CIA)
DD00-DD0F	56576-56591	Interface chip 2, NMI (6526 CIA)
D000-DFFF	53248-53294	Alternative: character set
E000-FFFF	57344-65535	ROM: operating system
E000-FFFF	57344-65535	Alternative: RAM

CIA 1 (IRQ)   (6526)   Commodore 64

Paddle SEL A . B			Joystick 0 R . L . D . U		
$DC00	Keyboard Row Select (inverted)			PRA	56320
$DC01	Keyboard Column Read	Joystick 1		PRB	56321
$DC02	$FF — All Output			DDRA	56322
$DC03	$00 — All Input			DDRB	56323
$DC04	Timer A			TAL	56324
$DC05				TAH	56325
$DC06	Timer B			TBL	56326
$DC07				TBH	56327
$DC0D	Tape input		Timer Interr B A	IOR	56333
$DC0E		One shot	Out mode	Time PDL out / Timer A start	ORA	56334
$DC0F		One shot	Out mode	Time PBC out / Timer B start	CRB	56335

Figure C.9

CIA 2 (NMI)   (6526)   Commodore 64

Addr	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Reg	Dec
$DD00	Serial In	Clock In	Serial Out	Clock Out	ATN Out	RS-232 Out	Video	Black	PRA	56576
$DD01	DSR In	CTS In		DCD* In	RI* In	DTR Out	RTS Out	RS-232 In	PRB	56577
				Parallel User Port						
$DD02	In	In	Out	Out	Out	Out	Out	Out	DDRA	56578
				$3F						
$DD03	$06 For RS-232								DDRB	56579
$DD04	Timer A								TAL	56580
$DD05	Timer A								TAH	56581
$DD06	Timer B								TBL	56582
$DD06	Timer B								TBH	56583
$DD0D			RS-232 In			Timer B	Timer A		ICR	56589
$DD0E								Timer A Start	CRA	56590
$DD0F								Timer B Start	CRB	56591

Figure C.10

C64 Memory Map
6566 Video — Sprite Registers

Sprite 0	Sprite 7		Sprite 0	Sprite 7
D000	D00E	Position X	53248	53262
D001	D00F	Y	53249	53263
D027	D02E	Color	53287	53294

Sprite Bit Positions

	7	6	5	4	3	2	1	0	
D010	X-position high								53264
D015	Sprite Enable								53269
D017	Y-expand								53271
D01B	Background Priority								53275
D01C	Multicolor								53276
D01D	X-expand								53277
D01E	Interrupt: Sprite collisn								53278
D01F	Interrupt: Sprite/Backgrd coll								53279

Figure C.11

SID (6581) Commodore 64

V1	V2	V3		V1	V2	V3
D400	D407	D40E	Frequency L	54272	54279	54286
D401	D408	D40F	Frequency H	54273	54280	54287
D402	D409	D410	Pulse Width L	54274	54281	54288
D403	D40A	D411	0 0 0 0 H	54275	54289	
				54382		
D404	D40B	D412	Voice Type KEY / NSE PUL SAW TRI	54276	54283	54290
D405	D40C	D413	Attack Time 2ms–8sec / Decay Time 6ms–24sec	54277	54284	54291
D406	D40D	D414	Sustain level / Release time 6ms–24sec	54278	54285	54292

Voices
(write only)

D415	0 0 0 0 0 L	54293
D416	Filter Frequency H	54294
D417	Resonance / Filter voices EXT V3 V2 V1	54295
D418	Passband V3 OFF HI BD LO / Master Volume	54296

Filter & Volume
(write only)

D419	Paddle X	54297
D41A	Paddle Y	54298
D41B	Noise 3 (random)	54299
D41C	Envelope 3	54300

Sense
(read only)

Special voice features (TEST, RING MOD, SYNC) are
omitted from the above diagram.

Figure C.12

Commodore PLUS/4 "TED" Chip— Preliminary

At time of publication the Commodore 264 (alternatively called Plus/4) and a related machine, the Commodore 16, are not commercially available. Design details could change before commercial release.

On the prototype units, much of zero-page is the same as for VIC and Commodore 64; in particular, the Basic pointers (SOB, SOV, etc.) are the same.

Memory Map, Preliminary

Much of zero-page is the same as for the Commodore 64. Some differences, and other information:

Hex	Decimal	Description
0073–008A	115–138	(CHARGET not present)
0097	151	How many open files
0098	152	Input device, normally 0
0099	153	Output CMD device, normally 3
00AC	172	Current logical file
00AD	173	Current secondary address
00AE	174	Current device
00AF–00B0	175–176	Pointer to file name
00C8–00C9	200–201	Pointer to screen line
00CA	202	Position of cursor on above line
00CD	205	Row where cursor lives
00EF	239	Number of characters in keyboard buffer
0314–0315	788–789	IRQ vector (CE0E)
0316–0317	790–791	Break interrupt vector (F44B)
0318–0319	792–793	OPEN vector (EF53)

(Most other vectors are similar to the C64, but are two locations lower)

0500–0502	1280–1282	USR program jump
0509–0512	1289–1298	Logical file table
0513–051C	1299–1308	Device number table
051D–0526	1309–1318	Secondary address table
0527–0530	1319–1328	Keyboard buffer
0800–0BE7	2048–3047	Color memory
0C00–0FE7	3072–4071	Screen memory

Address		Address
FF00	TIMERS — T1 L	65280
FF01	T1 H	65281
FF02	T2 L	65282
FF03	T2 H	65283
FF04	T3 L	65284
FF05	T3 H	65285
FF06	TEST ECM BMM BLANK ROWS Y-ADJUST	65286
FF07	RUS OFF PAL FREEZE MCM COLUMNS X-ADJUST	65287
FF08	KEYBOARD LATCH	65288
FF09	IRQ FLAG:T3 T2 T1 LP RAST	65289
FF0A	IER: T3 T2 T1 LP RAST	65290
FF0B	RC	65291
FF0C		65292
FF0D	CUR	65293
FF0E	SOUND-VOICE 1	65294
FF0F	VOICE 2	65295
FF10	VOICE 2 HI	65296
FF11	SOUND SELECT VOLUME	65297
FF12	BMB RBANK VOICE 1 HI	65298
FF13	CHARACTER BASE SCLOCK STATUS	65299
FF14	VIDEO MATRIX	65300
FF15	LUMINANCE COLOR 0	65301
FF16	1	65302
FF17	2	65303
FF18	BACKGROUND COLORS 3	65304
FF19	4	65305
FF1A		65306
FF1B	BRE	65307
FF1C		65308
FF1D	VL	65309
FF1E	H	65310
FF1F	BL VSUB	65311
FF3E	ROM SELECT	65342
FF3F	RAM SELECT	65343

Figure C.13

```
1000-FFFF  4096-65535    BASIC RAM memory
8000-FFFF  32768-65535   ROM: BASIC
FF00-FF3F  65280-65343   TED I/O control chip
```

B Series (B-128, CBM-256, etc.)
The Great Zero-Page Hunt

Zero page has a different meaning on the B series. There are several zero pages. Usually, you'll want to use values from bank 15 (the ROM bank, where system variables are kept); but if you are writing programs that will reside in a different bank, you'll have all of zero page (except locations 0 and 1) completely at your disposal.

If you need space in bank 15 zero page, you'll need to do some looking around. Addresses $E6 to $FF are not used by the system. Locations $20 to $2B and $64 to $6E are work areas available for temporary use.

Most zero-page locations may be copied to another part of memory so that their original contents can be restored after use. The programmer should take great care, however, in modifying the following locations, which are critical within the operating system or BASIC: $1A, $1D to $1F, $2D to $41, $43, $5B, $78, $85–87, $9E to $AB, $C0 to $E5.

Memory Map

The following information applies to B systems released after April 1983, which contain a revised machine language monitor. (If POKE 6, 0: SYS 6 doesn't bring in a monitor display complete with a "period" prompt, you have an incompatible version.)

Notable features as compared to previous Commodore products include:

—CHRGOT is no longer in RAM. Wedge-type coding must be inserted at links $029E and $02A0, which is likely to make the job easier.

—BASIC vectors have "split." Now, for example, there are discrete "start of variables" and "end of variables," distinct from end-of-BASIC and start-of-arrays. Three-byte vectors (including bank number) are not uncommon.

—The "jump table" at top of memory is still accessible and reasonably consistent with previous Commodore products.

—Simple machine language programs will fit into the spare 1K of ROM at $0400 to $07FF without trouble. Large programs must be implemented

either by plug-in memory (RAM or ROM) in bank 15 or by being placed into another bank (preferably bank 3). Supplementary code will be needed to make all the coding components fit.

The following map contains BASIC addresses specific to the B256/80; references to banks 0 to 4 are also specific to that machine. Most of the map is of general usage, however.

ALL BANKS:

0000	0	6509 execution register
0001	1	6509 indirection register

BANK 0: Unused.
BANK 1:

0002-F000	2-61439	BASIC program (text) RAM
FA5E-FB00	61440-64512	Input buffer area

BANK 2:
B256:

0002-FFFF	2-65535	BASIC arrays in RAM

B128:

0002-FFFF	2-65535	BASIC variables, arrays and strings
		Key definitions

BANK 3: (B256 only)

0002-7FFF	2-32767	Unused RAM.
8000-FFFF	32768-65535	BASIC variables in RAM

BANK 4: (B256 only)

0002-FBFF	2-64511	BASIC strings (top down) in RAM
FC00-FCFF	64512-64767	Unused RAM (descriptors?)
FD00-FFFF	64768-65535	Current KEY definitions

BANKS 5 to 14: Unused.
BANK 15:

0002-0004	2-4	USR jump
0005-0008	5-8	TI$ output elements: H,M,S,T
0009-000B	9-11	Print Using format pointer
000C	12	Search character
000D	13	Scan-between-quotes Flag
000E	14	Input point; number of subscripts
000F	15	Catalog line counter
0010	16	Default DIM flag
0011	17	Type: 255 = string, 0 = integer

0012	18	Type: 128 = integer, 0 = floating point
0013	19	Crunch flag
0014	20	Subscript index
0015	21	Input = 0; get = 64; read = 152.
0016–0019	22–25	Disk status work values
001A	26	Current I/O device for prompt suppress
001B–001C	27–28	Integer value
001D–001F	29–31	Descriptor stack pointers
0020–002B	32–43	Miscellaneous work pointers
002D–002E	45–46	Start-of-BASIC pointer
002F–0030	47–48	End-of-BASIC pointer
0031–0032	49–50	Start-of-Variables pointer
0033–0034	51–52	End-of-Variables pointer
0035–0036	53–54	Start-of-Arrays pointer
0037–0038	55–56	End-of-Arrays pointer
0039–003A	57–58	Variable work pointer
003B–003C	59–60	Bottom-of-Strings pointer
003D–003E	61–62	Utility string pointer
003F–0041	63–65	Top of string memory pointer
0042–0043	66–67	Current BASIC line number
0044–0045	68–69	Old BASIC line number
0046–0047	70–71	Old BASIC text pointer
0049–004A	73–74	Data line number
004B–004C	75–76	Data text pointer
004D–004E	77–78	Input pointer
004F–0050	79–80	Variable name
0051–0053	81–83	Variable address
0054–0056	84–86	For-loop pointer
0057–0058	87–88	Text pointer save
005A	90	Comparison symbol accumulator
005B–005D	91–92	Function location
005E–0060	94–96	Working string vector
0061–0063	97–99	Function jump code
0064–006E	100–110	Work pointers, values
006F	111	Exponent sign
0070	112	Acum string prefix
0071	113	Acum#1: exponent
0072–0075	114–117	Acum#1: mantissa
0076	118	Acum#1: sign

0077	119	Series evaluation constant pointer
0078	120	Acum#1 hi order (overflow)
0079-007E	121-126	Accum#2
007F	127	Sign comparison, Acc#1 versus #2
0080	128	Acc#1 low-order (rounding)
0081-0084	129-132	Series, work pointers
0085-0087	133-135	BASIC text pointer
0088-0089	136-137	Input pointer
008B-008E	139-142	DOS parser work values
008F	143	Error type number
0090-0092	144-146	Pointer to file name
0093-0095	147-149	Pointer: tape buffer, scrolling
0096-0098B	150-152	Load end address/end of program
0099-009B	153-155	I/O start address
009C	156	Status word ST
009D	157	File name length
009E	158	Current logical file
009F	159	Current device
00A0	160	Current secondary address
00A1	161	Input device, normally 0
00A2	162	Output CMD device, normally 3
00A6-00A8	166-168	INBUF
00A9	169	Keyswitch PIA: stop key, etc.
00AA	170	IEEE deferred flag
00AB	171	IEEE deferred character
00AC-00AD	172-173	Segment transfer routine vector
00AE-00B3	174-179	Monitor register save
00B4	180	Monitor stack pointer save
00B5	181	Monitor bank number save
00B7-00B8	183-184	Monitor IRQ save/pointer
00B9-00BA	185-186	Monitor memory pointer
00BB-00BC	187-188	Monitor secondary pointer
00BD	189	Monitor counter
00BE	190	Monitor miscellaneous byte
00BF	191	Monitor device number
00C0-00C1	192-193	Programmable key table address
00C2-00C3	194-195	Programmable key address
00C4-00C7	196-199	Pointers to change programmable key table

00C8–00C9	200–201	Pointer to screen line
00CA	202	Screen line number
00CB	203	Position of cursor on line
00CC	204	0 = text mode, else graphics mode
00CD	205	Keypress variable
00CE	206	Old cursor column
00CF	207	Old cursor row
00D0	208	New character flag
00D1	209	Number of keys in keyboard buffer
00D2	210	Quotes flag
00D3	211	Inert key counter
00D4	212	Cursor type flag
00D5	213	Screen line length
00D6	214	Number of keys in "key" buffer
00D7	215	Key repeat delay
00D8	216	Key repeat speed
00D9–00DA	217–218	Temporary variables
00DB	219	Current output character
00DC	220	Top line of current screen
00DD	221	Bottom line of screen
00DE	222	Left edge of current screen
00DF	223	Right edge of screen
00E0	224	Keys: 255 = none; 127 = key, 111 = shift
00E1	225	Key pressed: 255 = no key
00E2–00E5	226–229	Line Wrap Bits
0100	256	Hex to binary staging area
0100–010A	256–266	Numeric to ASCII work area
0100–01FE	256–510	Stack area
01FF	511	Stack pointer save location
0200–020F	512–527	File name area
0210–0226	528–550	Disk command work area
0255–0256	597	Miscellaneous work values for WAIT, etc
0257	599	"Bank" value
0258	600	Output logical file (CMD)
0259	601	Sign of TAN
025A–025D	602–605	Pickup subroutine; miscellaneous work values
025E–0276	606–630	PRINT USING working variables

0280-0281	640-641	Error routine link [8555]
0282-0283	642-643	Warm start link [85CD]
0284-0285	644-645	Crunch token link [88C2]
0286-0287	646-647	List link [89F4]
0288-0289	648-649	Command dispatch link [8754]
028A-028B	650-651	Token evaluate link [96B1]
028C-028D	652-653	Expression eval link [95C4]
028E-028F	654-655	CHRGOT link [BA2C]
0290-0291	656-657	CHRGET vector [BA32]
0292-0293	658-659	Float-fixed vector [BA1E]
0294-0295	660-661	Fixed-Float vector [9D39]
0296-0297	662-663	Error trap vector
0298-0299	664-665	Error line number
029A-029B	666-667	Error exit pointer
029C	668	Stack pointer save
029D-029F	669-671	Temporary TRAP, DISPOSE bytes
02A0-02A5	672-677	Temporary INSTR$ bytes
02A6-02A7	678-679	Bank offset
0300-0301	768-769	IRQ vector [FBE9]
0302-0303	770-771	BRK vector [EE21]
0304-0305	772-773	NMI vector [FCAA]
0306-0307	774-775	OPEN vector [F6BF]
0308-0309	776-777	CLOSE vector [F5ED]
030A-030B	778-779	Connect-input vector [F549]
030C-030D	780-781	Connect-output vector [F5A3]
030E-030F	782-783	Restore default I/O vector [F6A6]
0310-0311	784-785	Input vector [F49C]
0312-0313	786-787	Output vector [F4EE]
0314-0315	788-789	Stop key test vector [F96B]
0316-0317	790-791	GET vector [F43D]
0318-0319	792-793	Abort all files vector [F67F]
031A-031B	794-795	Load vector [F746]
031C-031D	796-797	Save vector [F84C]
031E-031F	798-799	Monitor command vector EE7?]
0320-0321	800-801	Keyboard control vector [E01F]
0322-0323	802-803	Print control vector [E01F]
0324-0325	804-805	IEEE send LSA vector F274]
0326-0327	806-807	IEEE send TSA vector F280]

0328-0329	808-809	IEEE receive byte vector F30A]
032A-032B	810-811	IEEE send character vector F297]
032C-032D	812-813	IEEE send untalk vector F2AB]
032E-032F	814-815	IEEE send unlisten vector F2AF]
0330-0331	816-817	IEEE send listen vector F234]
0332-0333	818-819	IEEE send talk vector [F230]
0334-033D	820-829	File logical addresses table
033E-0347	830-839	File device table
0348-0351	840-849	File secondary address table
0352-0354	850-852	Bottom of system memory
0355-0357	853-855	Top of system memory
0358-035A	856-858	Bottom of user memory
035B-035D	859-861	Top of user memory
035E	862	IEEE timeout; 0 = enabled
035F	863	0 = load; 128 = verify
0360	864	Number of open files
0361	865	Message mode byte
0363-0366	867-870	Miscellaneous register save bytes
0369	873	Timer toggle
036A-036B	874-875	Cassette vector (dead end)
036F-0371	879-881	Relocation start address
0375	885	Cassette motor flag (unused)
0376-0377	886-887	RS-232 control, command
037A	890	RS-232 status
037B	891	RS-232 handshake input
037C	892	RS-232 input pointer
037D	893	RS-232 arrival pointer
0380-0381	896-897	Top of memory pointer
0382	898	Bank byte
0383	899	RVS flag
0384	900	Current line length
0385	901	Temporary output character save
0386	902	0 = normal, 255 = auto insert
0387	903	0 = scrolling, 255 = no scroll
0388	904	Miscellaneous work byte for screen

0389	905	Index to programmed key
038A	906	Scroll mode flag
038B	907	Bell mode flag
038C	908	Indirect bank save
038D–03A0	909–928	Lengths of 'key' words
03A1–03AA	929–938	Bit mapped tab stops
03AB–03B4	939–948	Keyboard input buffer
03B5–03B6	949–950	'Key' word link [E91B]
03F8–03F9	1016–1017	Restart vector
03FA–03FB	1018–1019	Restart test mask
0400–07FF	1024–2047	Free RAM (reserved for DOS)
0800–0FFF	2048–4095	Reserved for plug in RAM
1000–1FFF	4096–8191	Reserved for plug in DOS ROM
2000–7FFF	8192–23767	Reserved for cartridges
8000–BFFF	32768–49151	BASIC ROM
C000–CFFF	49152–53247	Unused
D000–D7CF	53248–55247	Screen RAM
D800–D801	55296–55297	Video controller 6545
DA00–DA1C	55808–55836	Sound interface device 6581
DB00–DB0F	56064–56079	Complex interface adaptor 6526
DC00–DC0F	56320–56335	Complex interface adaptor 6526
DD00–DD03	56576–56579	Asynchronous communications IA 6551
DE00–DE07	56832–56839	Tri Port Interface Adaptor 6525
DF00–DF07	57088–57095	Tri Port Interface Adaptor 6525
E000–FFFF	57344–65535	Kernal ROM

The above table shows contents for the link and vector addresses at $0280 to $0295; these are taken from a recent B-128.

6545 CRT Controller

D800 55296	D801 55297	Typical Value (Decimal)
0	Horizontal Total	108 or 126 or 127
1	Horizontal Char Displayed	80
2	Horizontal Sync Position	83 or 98 or 96
3	V Sync Width H	15 or 10
4	Vertical Total	25 or 31 or 38
5	Vert Total Adjust	3 or 6 or 1
6	Vertical Displayed	25
7	Vert Sync Position	25 or 28 or 30
8	Mode	0
9	Scan Lines	13 or 7
10	Cursor Start	96 (blink) or 0 or 6 (underline)
11	Cursor End	13 or 7
12	Display Address H	0
13	L	0
14	Cursor Address H	Varies
15	L	Varies
16	Light Pen In H	0
17	L	0

Most Registers are Write Only   14/15 are Read/Write
16/17 are Read Only
Registers 10, 14 and 15 change as the cursor moves

Figure C.14

6525 Tri Port

DE00	NRFD	NDAC	EOI	DAV	ATN	RFN			56832
DE01	Sense	Cassette Motor	Out	ARB	Network Rx	Tx	SRQ	IFC	56833
DE02									56834
DE03	Data Direction Register For DE00								56835
DE04	Data Direction Register For DE01								56836
DE05	IRQ			ACIA	IP	ALM	IEEE	PWR	56837
DE06	CB		CA Graphics				IRQ Stack On		56838
DE07	Active Interrupt Register								56839

6525 Tri Port 2

DF00	Keyboard		57088
DF01	Select		57089
DF02	CRT Mode	Keyboard Read	57090
DF03	Data Direction Register for DF00 (out)		57091
DF04	Data Direction Register for DF01 (out)		57092
DF05	Data Direction Register for DF02 (in)		57093
DF06	Unused		57094

Figure C.15

Commodore 64: ROM Detail

This type of ROM memory map is intended primarily for users who want to "browse" through the inner logic of the computer. It allows a user to disassemble an area of interest, to see why the computer behaves in a certain way. With the use of this map, the user will be able to identify subroutines that are called by the coding under study.

I recommend against using the ROM subroutines as part of your own programs. They often don't do precisely what you want. They change locations when you move to a different machine. With rare exceptions, you can probably write better coding to do the job yourself. Stick with the kernal jump table: especially $FFD2 to output; $FFE4 to get input; $FFE1 to check the RUN/STOP key; $FFC6 and $FFC9 to switch input and output respectively; and $FFCC to restore normal input/output. They are the same on all Commodore computers.

    A000:   ROM control vectors
    A00C: Keyword action vectors

A052:	Function vectors
A080:	Operator vectors
A09E:	Keywords
A19E:	Error messages
A328:	Error message vectors
A365:	Miscellaneous messages
A38A:	Scan stack for FOR/GOSUB
A3B8:	Move memory
A3FB:	Check stack depth
A408:	Check memory space
A435:	Print "out of memory"
A437:	Error routine
A469:	BREAK entry
A474:	Print "ready."
A480:	Ready for BASIC
A49C:	Handle new line
A533:	Re-chain lines
A560:	Receive input line
A579:	Crunch tokens
A613:	Find BASIC line
A642:	Perform [NEW]
A65E:	Perform [CLR]
A68E:	Back up text pointer
A69C:	Perform [LIST]
A742:	Perform [FOR]
A7ED:	Execute statement
A81D:	Perform [RESTORE]
A82C:	Break
A82F:	Perform [STOP]
A831:	Perform [END]
A857:	Perform [CONT]
A871:	Perform [RUN]
A883:	Perform [GOSUB]
A8A0:	Perform [GOTO]
A8D2:	Perform [RETURN]
A8F8:	Perform [DATA]
A906:	Scan for next statement
A928:	Perform [IF]
A93B:	Perform [REM]
A94B:	Perform [ON]
A96B:	Get fixed point number
A9A5:	Perform [LET]
AA80:	Perform [PRINT#]

```
AA86: Perform [CMD]
AAA0: Perform [PRINT]
AB1E: Print string from (Y.A)
AB3B: Print format character
AB4D: Bad input routine
AB7B: Perform [GET]
ABA5: Perform [INPUT#]
ABBF: Perform [INPUT]
ABF9: Prompt and input
AC06: Perform [READ]
ACFC: Input error messages
AD1E: Perform [NEXT]
AD78: Type match check
AD9E: Evaluate expression
AEA8: Constant-pi
AEF1: Evaluate within brackets
AEF7: Check for " ) "
AEFF: Check for comma
AF08: Syntax error
AF14: Check range
AF28: Search for variable
AFA7: Set up FN reference
AFE6: Evaluate [OR]
AFE9: Evaluate [AND]
B016: Compare
B081: Perform [DIM]
B08B: Locate variable
B113: Check alphabetic
B11D: Create variable
B194: Array pointer subroutine
B1A5: Value 32768
B1B2: Float-fixed
B1D1: Set up array
B245: Print "bad subscript"
B248: Print "illegal quantity"
B34C: Compute array size
B37D: Evaluate [FRE]
B391: Fixed-float
B39E: Evaluate [POS]
B3A6: Check direct
B3B3: Perform [DEF]
B3E1: Check fn syntax
B3F4: Evaluate [FN]
```

B465: Evaluate [STR$]
B475: Calculate string vector
B487: Set up string
B4F4: Make room for string
B526: Garbage collection
B5BD: Check salvageability
B606: Collect string
B63D: Concatenate
B67A: Build string to memory
B6A3: Discard unwanted string
B6DB: Clean descriptor stack
B6EC: Evaluate [CHR$]
B700: Evaluate [LEFT$]
B72C: Evaluate [RIGHT$]
B737: Evaluate [MID$]
B761: Pull string parameters
B77C: Evaluate [LEN]
B782: Exit string-mode
B78B: Evaluate [ASC]
B79B: Input byte parameter
B7AD: Evaluate [VAL]
B7EB: Parameters for POKE/WAIT
B7F7: Float-fixed
B80D: Evaluate [PEEK]
B824: Perform [POKE]
B82D: Perform [WAIT]
B849: Add 0.5
B850: Subtract-from
B853: Evaluate [subtract]
B86A: Evaluate [add]
B947: Complement FAC (floating accumulator)#1
B97E: Print "overflow"
B983: Multiply by zero byte
B9EA: Evaluate [LOG]
BA2B: Evaluate [multiply]
BA59: Multiply-a-bit
BA8C: Memory to FAC#2
BAB7: Adjust FAC#1 and FAC#2
BAD4: Underflow/overflow
BAE2: Multiply by 10
BAF9: +10 in floating point
BAFE: Divide by 10
BB12: Evaluate [divide]

`BBA2:`   Memory to `FAC#1`
`BBC7:`   `FAC#1` to memory
`BBFC:`   `FAC#2` to `FAC#1`
`BC0C:`   `FAC#1` to `FAC#2`
`BC1B:`   Round `FAC#1`
`BC2B:`   Get sign
`BC39:`   Evaluate `[SGN]`
`BC58:`   Evaluate `[ABS]`
`BC5B:`   Compare `FAC#1` to memory
`BC9B:`   Float-fixed
`BCCC:`   Evaluate `[INT]`
`BCF3:`   String to `FAC`
`BD7E:`   Get ASCII digit
`BDC2:`   Print `"IN.."`
`BDCD:`   Print line number
`BDDD:`   Float to ASCII
`BF16:`   Decimal constants
`BF3A:`   `TI` constants
`BF71:`   Evaluate `[SQR]`
`BF7B:`   Evaluate `[power]`
`BFB4:`   Evaluate `[negative]`
`BFED:`   Evaluate `[EXP]`
`E043:`   Series evaluation 1
`E059:`   Series evaluation 2
`E097:`   Evaluate `[RND]`
`E0F9:`   Kernal calls with error checking
`E12A:`   Perform `[SYS]`
`E156:`   Perform `[SAVE]`
`E165:`   Perform `[VERIFY]`
`E168:`   Perform `[LOAD]`
`E1BE:`   Perform `[OPEN]`
`E1C7:`   Perform `[CLOSE]`
`E1D4:`   Parameters for `LOAD/SAVE`
`E206:`   Check default parameters
`E20E:`   Check for comma
`E219:`   Parameters for open/close
`E264:`   Evaluate `[COS]`
`E26B:`   Evaluate `[SIN]`
`E2b4:`   Evaluate `[TAN]`
`E30E:`   Evaluate `[ATN]`
`E37B:`   Warm restart
`E394:`   Initialize
`E3A2:`   `CHRGET` for zero page

E3BF: Initialize BASIC
E447: Vectors for $300
E453: Initialize vectors
E45F: Power-up message
E500: Get I/O address
E505: Get screen size
E50A: Put/get row/column
E518: Initialize I/O
E544: Clear screen
E566: Home cursor
E56C: Set screen pointers
E5A0: Set I/O defaults
E5B4: Input from keyboard
E632: Input from screen
E684: Quote test
E691: Set up screen print
E6B6: Advance cursor
E6ED: Retreat cursor
E701: Back into previous line
E716: Output to screen
E87C: Go to next line
E891: Perform (return)
E8A1: Check line decrement
E8B3: Check line increment
E8CB: Set color code
E8DA: Color code table
E8EA: Scroll screen
E965: Open space on screen
E9C8: Move a screen line
E9E0: Synchronize color transfer
E9F0: Set start-of-line
E9FF: Clear screen line
EA13: Print to screen
EA24: Synchronize color pointer
EA31: Interrupt-clock, etc.
EA87: Read keyboard
EB79: Keyboard select vectors
EB81: Keyboard 1-unshifted
EBC2: Keyboard 2-shifted
EC03: Keyboard 3-"Commodore" shift
EC44: Graphics/text contrl
EC4F: Set graphics/text mode
EC78: Keyboard 4

ECB9:   Video chip setup
ECE7:   Shift/run equivalent
ECF0:   Screen ln address low
ED09:   Send "talk" to serial bus
ED0C: Send "listen" to serial bus
ED40:   Send to serial bus
EDB2:   Serial timeout
EDB9:   Send listen S A
EDBE: Clear A T N
EDC7:   Send talk S A
EDCC: Wait for clock
EDDD: Send serial deferred
EDEF: Send "untalk" to serial bus
EDFE: Send "unlisten" to serial bus
EE13:   Receive from serial bus
EE85:   Serial clock on
EE8E: Serial clock off
EE97:   Serial output "1"
EEA0:   Serial output "0"
EEA9:   Get serial in and clock signals
EEB3:   Delay 1 millisecond
EEBB: RS-232 send
EF06:   Send new RS-232 byte
EF2E: No-D S R error
EF31:   No-C T S error
EF3B: Disable timer
EF4A: Compute bit count
EF59:   RS-232 receive
EF7E: Set up to receive
EFC5:   Receive parity error
EFCA: Recieve overflow
EFCD: Receive break
EFD0:   Framing error
EFE1:   Submit to RS-232
F00D: No-D S R error
F017:   Send to RS-232 buffer
F04D: Input from RS-232
F086:   Get from RS-232
F0A4:   Check serial bus idle
F0BD: Messages
F12B: Print if direct
F13E: Get . . .
F14E: . . . from RS-232

F157 :   Input
F199 :   Get: tape/serial/RS-232
F1CA :   Output . . .
F1DD :   . . . to tape
F20E :   Set input device
F250 :   Set output device
F291 :   Close file
F30F :   Find file
F31F :   Set file values
F32F :   Abort all files
F333 :   Restore default I/O
F34A :   Do file open
F3D5 :   Send S A
F409 :   Open RS-232
F49E :   Load program
F5AF :   Print "searching"
F5C1 :   Print filename
F5D2 :   Print "loading/verifying"
F5DD :   Save program
F68F :   Print "saving"
F69B :   Bump clock
F6BC :   Log P I A key reading
F6DD :   Get time
F6E4 :   Set time
F6ED :   Check stop key
F6FB :   Output error messages
F72D :   Find any tape header
F76A :   Write tape header
F7D0 :   Get buffer address
F7D7 :   Set buffer start/end pointers
F7EA :   Find specific header
F80D :   Bump tape pointer
F817 :   Print "press play   "
F82E :   Check tape status
F838 :   Print "press record . . ."
F841 :   Initiate tape read
F864 :   Initiate tape write
F875 :   Common tape code
F8D0 :   Check tape stop
F8E2 :   Set read timing
F92C :   Read tape bits
FA60 :   Store tape characters
FB8E :   Reset pointer

```
FB97:  New character setup
FBA6:  Send transition to tape
FBC8:  Write data to tape
FBCD:  IRQ entry point
FC57:  Write tape leader
FC93:  Restore normal IRQ
FCB8:  Set IRQ vector
FCCA:  Kill tape motor
FCD1:  Check R/W pointer
FCDB:  Bump R/W pointer
FCE2:  Power reset entry
FD02:  Check 8-ROM
FD10:  8-ROM mask
FD15:  Kernal reset
FD1A:  Kernal move
FD30:  Vectors
FD50:  Initialize system constants
FD9B:  IRQ vectors
FDA3:  Initialize I/O
FDDD:  Enable timer
FDF9:  Save filename data
FE00:  Save file details
FE07:  Get status
FE18:  Flag status
FE1C:  Set status
FE21:  Set timeout
FE25:  Read/set top of memory
FE27:  Read top of memory
FE2D:  Set top of memory
FE34:  Read/set bottom of memory
FE43:  NMI entry
FE66:  Warm start
FEB6:  Reset IRQ and exit
FEBC:  Interrupt exit
FEC2:  RS-232 timing table
FED6:  NMI RS-232 in
FF07:  NMI RS-232 out
FF43:  Fake IRQ
FF48:  IRQ entry
FF81:  Jumbo jump table
FFFA:  Hardware vectors
```

D

Character Sets

Superchart

The "superchart" shows the PET character sets. A byte may have any of several meanings, depending on how it is used. The chart is constructed to reflect this. "ASCII" is PET ASCII; these are the characters as they would be input or printed. "Screen" is the Commodore screen codes, as they would be used in screen memory—POKEing to or PEEKing from the screen would yield these codes. Notice that the numeric character set is the same for both screen and PET ASCII.

Within a program, the code changes again "BASIC" shows these codes; they are similar to ASCII in the range $20 to $5F.

Machine language op codes are included for the sake of convenience and completeness

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
0	00		@	end-line	BRK	0
1	01		A		ORA(I,X)	1
2	02		B			2
3	03		C			3
4	04		D			4
5	05	white	E		ORA Z	5
6	06		F		ASL Z	6
7	07	bell	G			7
8	08	lock	H		PHP	8
9	09	unlock	I		ORA #	9
10	0A		J		ASL A	10
11	0B		K			11
12	0C		L			12
13	0D	car ret	M		ORA	13
14	0E	text	N		ASL	14
15	0F	top	O			15
16	10		P		BPL	16
17	11	cur down	Q		ORA(I),Y	17
18	12	reverse	R			18
19	13	cur home	S			19
20	14	delete	T			20
21	15	del line	U		ORA Z,X	21
22	16	ers begin	V		ASL Z,X	22
23	17		W			23
24	18		X		CLC	24
25	19	scr up	Y		ORA Y	25
26	1A		Z			26
27	1B		[			27
28	1C	red	\			28
29	1D	cur right	]		ORA X	29

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
30	1E	green	↑		ASL X	30
31	1F	blue	←			31
32	20	space	space	space	JSR	32
33	21	!	!	!	AND(I,X)	33
34	22	"	"	"		34
35	23	#	#	#		35
36	24	$	$	$	BIT Z	36
37	25	%	%	%	AND Z	37
38	26	&	&	&	ROL Z	38
39	27	'	'	'		39
40	28	(	(	(	PLB	40
41	29	)	)	)	AND #	41
42	2A	*	*	*	ROL A	42
43	2B	+	+	+		43
44	2C	'	'	'	BIT	44
45	2D	-	-	-	AND	45
46	2E		.		ROL	46
47	2F	/	/	/		47
48	30	0	0	0	BMI	48
49	31	1	1	1	AND(I),Y	49
50	32	2	2	2		50
51	33	3	3	3		51
52	34	4	4	4		52
53	35	5	5	5	AND Z,X	53
54	36	6	6	6	ROL Z,X	54
55	37	7	7	7		55
56	38	8	8	8	SEC	56
57	39	9	9	9	AND Y	57
58	3A		.		CLI	58
59	3B	,	,	,		59
60	3C	<	<	<		60
61	30	=	=	=	AND X	61
62	3E	>	>	>	ROL X	62
63	3F	?	?	?		63
64	40	@	▬	@	RTI	64
65	41	A	♠ a	A	EOR(I,X)	65
66	42	B	▯. b	B		66
67	43	C	▬. c	C		67
68	44	D	▭. d	D		68
69	45	E	▬. e	E	EOR Z	69
70	46	F	▭ f	F	LSR Z	70
71	47	G	▯ g	G		71
72	48	H	▯. h	H	PHA	72
73	49	I	◣. i	I	EOR #	73
74	4A	J	◥ j	J	LSR A	74
75	4B	K	◢, k	K		75
76	4C	L	◣, l	L	JMP	76
77	4D	M	◥, m	M	EOR	77
78	4E	N	◤, n	N	LSR	78

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
79	4F	O	⬛, o	O		79
80	50	P	⬛ p	P	BVC	80
81	51	Q	⬛, q	Q	EOR(I),Y	81
82	52	R	⬛, r	R		82
83	53	S	⬛, s	S		83
84	54	T	⬛, t	T		84
85	55	U	⬛, u	U	EOR, Z,X	85
86	56	V	⬛, v	V	LSR Z,X	86
87	57	W	⬛, w	W		87
88	58	X	⬛, x	X	CLI	88
89	59	Y	⬛, y	Y	EOR Y	89
90	5A	Z	⬛, z	Z		90
91	5B	[	⬛	[		91
92	5C	\	⬛	\		92
93	5D	]	⬛	]	EOR X	93
94	5E	↑	⬛, ⬛	↑	LSR X	94
95	5F	←	⬛, ⬛	←		95
96	60		⬛		RTS	96
97	61		⬛		ADC(I,X)	97
98	62		⬛			98
99	63		⬛			99
100	64		⬛			100
101	65		⬛		ADC Z	101
102	66		⬛		ROR Z	102
103	67		⬛			103
104	68		⬛		PLA	104
105	69		⬛, ⬛		ADC #	105
106	6A		⬛		ROR A	106
107	6B		⬛			107
108	6C		⬛		JMP(I)	108
109	6D		⬛		ADC	109
110	6E		⬛		ROR	110
111	6F		⬛			111
112	70		⬛		RVS	112
113	71		⬛		ADC(I),Y	113
114	72		⬛			114
115	73		⬛			115
116	74		⬛			116
117	75		⬛		ADC Z,X	117
118	76		⬛		ROR Z,X	118
119	77		⬛			119
120	78		⬛		SEI	120
121	79		⬛		ADC Y	121
122	7A		⬛ ✓			122
123	7B		⬛			123
124	7C		⬛			124
125	7D		⬛		ADC X	125
126	7E		⬛		ROR X	126
127	7F		⬛			127

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
128	80		r-((	END		128
129	81	orange	r-A	FOR	STA(I,X)	129
130	82		r-B	NEXT		130
131	83		r-C	DATA		131
132	84		r-D	INPUT#	STY Z	132
133	85		r-E	INPUT	STA Z	133
134	86		r-F	DIM	STX Z	134
135	87		r-G	READ		135
136	88		r-H	LET	DEY	136
137	89		r-I	GOTO		137
138	8A		r-J	RUN	TXA	138
139	8B		r-K	IF		139
140	8C		r-L	RESTORE	STY	140
141	8D	car ret	r-M	GOSUB	STA	141
142	8E	graphic	r-N	RETURN	STX	142
143	8F	bottom	r-O	REM		143
144	90	black	r-P	STOP	BCC	144
145	91	cur up	r-Q	ON	STA(I), Y	145
146	92	rvs off	r-R	WAIT		146
147	93	clear	r-S	LOAD		147
148	94	insert	r-T	SAVE	STY Z,X	148
149	95	ins line/br	r-U	VERIFY	STA Z,X	149
150	96	ers end/p	r-V	DEF	STX Z,Y	150
151	97	Gray 1	r-W	POKE		151
152	98	Gray 2	r-X	PRINT#	TYA	152
153	99	scr. down	r-Y	PRINT	STA Y	153
154	9A	L Blue	r-Z	CONT	TXS	154
155	9B	Gray 3	r-[	LIST		155
156	9C	magenta	r-\	CLR		156
157	9D	cur left	r-]	CMD	STA X	157
158	9E	yellow	r- ↑	SYS		158
159	9F	cyan	r-←	OPEN		159
160	A0		■	CLOSE	LDY #	160
161	A1		r-!	GET	LDA(I,X)	161
162	A2		r-"	NEW	LDX #	162
163	A3		r-#	TAB(		163
164	A4		r-$	TO	LDY Z	164
165	A5		r-%	FN	LDA Z	165
166	A6		r-&	SPC(	LDX Z	166
167	A7		r-'	THEN		167
168	A8		r-(	NOT	TAY	168
169	A9		r-)	STEP	LDA #	169
170	AA		r-*	+	TAX	170
171	AB		r- +	−		171
172	AC		r-,	*	LDY	172
173	AD		r--	/	LDA	173
174	AE		r-.	↑	LDX	174
175	AF		r-/	AND		175
176	B0		r-0	OR	BCS	176

DECIMAL	HEX	ASCII		SCREEN	BASIC	6502	DECIMAL
177	B1	▨		r-1	>	LDA(I),Y	177
178	B2	▨		r-2	=		178
179	B3	▨		r-3	<		179
180	B4	▯		r-4	SGN	LDY Z,X	180
181	B5	▯		r-5	INT	LDA Z,X	181
182	B6	▯		r-6	ABS	LDX Z,Y	182
183	B7	▭		r-7	USR		183
184	B8	▬		r-8	FRE	CLV	184
185	B9	▭		r-9	POS	LDA Y	185
186	BA	▯	☑	r-.	SQR	TSX	186
187	BB	▨		r-,	RND		187
188	BC	▮		r-<	LOG	LDY X	188
189	BD	▨		r- =	EXP	LDA X	189
190	BE	▮		r->	COS	LDX Y	190
191	BF	▮		r-?	SIN		191
192	C0	▬		TAN		CPY#	192
193	C1	▲	a	ATN		CMP(I),X	193
194	C2	▥	b	PEEK			194
195	C3	▬	c	LEN			195
196	C4	▤	d	STR$		CPY Z	196
197	C5	▧	e	VAL		CMP Z	197
198	C6	▬	f	ASC		DEC Z	198
199	C7	▥	g	CHR$			199
200	C8	▥	h	LEFT$		INY	200
201	C9	◣	i	RIGHT$		CMP #	201
202	CA	◥	j	MID$		DEX	202
203	CB	◪	k	GO			203
204	CC	▮	l	CONCAT		CPY	204
205	CD	◥	m	DOPEN		CMP	205
206	CE	◢	n	DCLOSE		DEC	206
207	CF	◰	o	RECORD			207
208	D0	◙	p	HEADER		BNE	208
209	D1	◖	q	COLLECT		CMP(I),Y	209
210	D2	◗	r	BACKUP			210
211	D3	♥	s	COPY			211
212	D4	▯	t	APPEND			212
213	D5	◪	u	DSAVE		CMP Z,X	213
214	D6	✕	v	DLOAD		DEC Z,X	214
215	D7	◉	w	CATALOG			215
216	D8	▨	x	RENAME		CLD	216
217	D9	▥	y	SCRATCH		CMP Y	217
218	DA	◆	z	DIRECTORY			218
219	DB	▦		⋮			219
220	DC	◧		⋮			220
221	DD	▮		⋮		CMP X	221
222	DE	◩	▨	⋮		DEC X	222
223	DF	◣	▨	↓			223
224	E0	▬		■		CPX #	224
225	E1	▐		▮		SBC(I),X	225

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
226	E2					226
227	E3					227
228	E4				CPX Z	228
229	E5				SBC Z	229
230	E6				INC Z	230
231	E7					231
232	E8				INX	232
233	E9				SBC #	233
234	EA				NOP	234
235	EB					235
236	EC				CPX	236
237	ED				SBC	237
238	EE				INC	238
239	EF					239
240	FO				BEQ	240
241	F1				SBC(I), Y	241
242	F2					242
243	F3					243
244	F4					244
245	F5				SBC Z,X	245
246	F6				INC Z,X	246
247	F7					247
248	F8				SED	248
249	F9				SBC Y	249
250	FA					250
251	FB					251
252	FC					252
253	FD				SBC X	253
254	FE				INC X	254
255	FF			π		255

Control Character Representations

NUL	Null	DLE	Data Link Escape (CC)
SOH	Start of Heading (CC)	DC1	Device Control 1
STX	Start of Text (CC)	DC2	Device Control 2
ETX	End of Text (CC)	DC3	Device Control 3
EOT	End of Transmission (CC)	DC4	Device Control 4
ENQ	Enquiry (CC)	NAK	Negative Acknowledge (CC)
ACK	Acknowledge (CC)	SYN	Synchronous Idle (CC)
BEL	Bell	ETB	End of Transmission Block
BS	Backspace (FE)		(CC)
HT	Horizontal Tabulation	CAN	Cancel
	(FE)	EM	End of Medium
LF	Line Feed (FE)	SUB	Substitute
VT	Vertical Tabulation (FE)	ESC	Escape
FF	Form Feed (FE)	FS	File Separator (IS)
→ CR	Carriage Return (FE)	GS	Group Separator (IS)
SO	Shift Out	RS	Record Separator (IS)
SI	Shift In	US	Unit Separator (IS)
		DEL	Delete

(CC)	Communication Control
(FE)	Format Effector
(IS)	Information Separator

Figure D.1

Special Graphic Characters

→ SP	Space	→ <	Less Than
→ !	Exclamation Point	→ =	Equals
→ "	Quotation Marks	→ >	Greater Than
→ #	Number Sign	→ ?	Question Mark
→ $	Dollar Sign	→ @	Commercial At
→ %	Percent	→ [	Opening Bracket
→ &	Ampersand	\	Reverse Slant
→ '	Apostrophe	→]	Closing Bracket
→ (	Opening Parenthesis	^	Circumflex
→)	Closing Parenthesis	_	Underline
→ *	Asterisk	`	Grave Accent
→ +	Plus	{	Opening Brace
→ ,	Comma	|	Vertical Line (This graphic
→ −	Hyphen (Minus)		is sometimes stylized to
→ .	Period (Decimal Point)		distinguish it from the un-
→ /	Slant		broken Logical OR which
→ :	Colon		is not an ASCII character)
→ ;	Semicolon	}	Closing Brace
		~	Tilde

Characters marked → correspond to the PET ASCII character set.

Figure D.2

ASCII

ASCII is the American Standard Code for Information Interchange. It is the standard for communications, and is often used with non-Commodore printers.

When a Commodore machine is in its graphic mode, its character set corresponds closely to ASCII. Numeric, upper case alphabetic, and punctuation characters are the same A few control characters, such as RETURN, also match. Commodore graphics have no counterpart in ASCII.

When the Commodore machine is switched to text mode, the character set diverges noticeably from ASCII. Numeric characters and much of the punctuation corresponds, but ASCII upper case alphabetic codes match the Commodore computer's lower case codes. Commodore's upper case alphabetics are now completely out of the ASCII range, since ASCII is a seven-bit code

As a result, Commodore's PET ASCII codes require conversion before transmission to a true ASCII device or communications line. This may be done with either hardware interfacing or with a program. Briefly, the procedure is:

1. If the Commodore character is below $3F, it may be transmitted directly to the ASCII facility.

2. If the Commodore character is between $40 and $5F, it should be logically ORed with $20 (or add decimal 32) before transmission to ASCII

3. If the Commodore character is between $C0 and $DF, it should be logically ANDed with $7F (or subtract decimal 128)before transmission to ASCII.

Equivalent rules can be derived to allow a Commodore computer to receive from ASCII. For either direction of transmission, some control characters may require special treatment.

First Hexadecimal Digit

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DCI	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	–	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	—	o	DEL

(Second Hexadecimal Digit — row labels at left)

ASCII code values.

E

Exercises for Alternative Commodore Machines

From Chapter 6:
VIC-20 (Unexpanded) Version

We write the BASIC program as follows:

```
100  V%=0
110  FOR J=1 TO 5
120  INPUT "VALUE";V%
130  SYS ++++
140  PRINT "TIMES TEN =";V%
150  NEXT J
```

Plan to start the machine language program at around $4097+127$, or 4224 (hexadecimal 1080). On that basis, we may change line 130 to SYS 4224. *Do not* try to run the program yet.

```
.A 1080  LDY  #$02
.A 1082  LDA  ($2D),Y
.A 1084  STA  $033C
.A 1087  STA  $033E
.A 108A  LDY  #$03
.A 108C  LDA  ($2D),Y
.A 108E  STA  $033D
.A 1091  STA  $033F
.A 1094  ASL  $033D
.A 1097  ROL  $033C
.A 109A  ASL  $033D
.A 109D  ROL  $033C
.A 10A0  CLC
.A 10A1  LDA  $033D
.A 10A4  ADC  $033F
.A 10A7  STA  $033D
.A 10AA  LDA  $033C
.A 10AD  ADC  $033E
.A 10B0  STA  $033C
.A 10B3  ASL  $033D
.A 10B6  ROL  $033C
.A 10B9  LDY  #$02
.A 10BB  LDA  $033C
.A 10BE  STA  ($2D),Y
.A 10C0  LDY  #$03
.A 10C2  LDA  $033D
.A 10C5  STA  ($2D),Y
.A 10C7  RTS
```

To change the start-of-variables pointer to a location above the machine language program, display the `SOV` pointer with `.M 002D 002E` and change the pointer to

`.:002D C8 10 .. .. .. .. .. .. ..`

PET/CBM Version

We write the BASIC program as follows:

```
100  V%=0
110  FOR J=1 TO 5
120  INPUT "VALUE";V%
130  SYS ++++
140  PRINT "TIMES TEN =";V%
150  NEXT J
```

Plan to start the machine language program at around 1025 + 127, or 1152 (hexadecimal 480). On that basis, we may change line 130 to SYS 1152. *Do not* try to run the program yet.

```
.A 0480   LDY   #$02
.A 0482   LDA   ($2A),Y
.A 0484   STA   $033C
.A 0487   STA   $033E
.A 048A   LDY   #$03
.A 048C   LDA   ($2A),Y
.A 048E   STA   $033D
.A 0491   STA   $033F
.A 0494   ASL   $033D
.A 0497   ROL   $033C
.A 049A   ASL   $033D
.A 049D   ROL   $033C
.A 04A0   CLC
.A 04A1   LDA   $033D
.A 04A4   ADC   $033F ·
.A 04A7   STA   $033D
.A 04AA   LDA   $033C
.A 04AD   ADC   $033E
.A 04B0   STA   $033C
.A 04B3   ASL   $033D
.A 04B6   ROL   $033C
.A 04B9   LDY   #$02
.A 04BB   LDA   $033C
.A 04BE   STA   ($2A),Y
```

```
.A 04C0 LDY #$03
.A 04C2 LDA $033D
.A 04C5 STA ($2A),Y
.A 04C7 RTS
```

To change the start-of-variables pointer to a location above the machine language program, display the SOV pointer with .M 002A 002B and change the pointer to

```
.:002A C8 04 .. .. .. .. .. .. ..
```

From Chapter 7:
An Interrupt Project
VIC-20 (Unexpanded) Version

The only difference with the VIC-20 is that the screen is located at $1E00:

```
.A 033C LDA $91
.A 033E STA $1E00
.A 0341 JMP ($03A0)
```

To place the link address into $03A0/1:

```
.A 0344 LDA $0314
.A 0347 STA $03A0
.A 034A LDA $0315
.A 034D STA $03A1
```

To fire up the program:

```
.A 0350 SEI
.A 0351 LDA #$3C
.A 0353 STA $0314
.A 0356 LDA #$03
.A 0358 STA $0315
.A 035B CLI
.A 035C RTS
```

To restore the original interrupt:

```
.A 035D SEI
.A 035E LDA $03A0
.A 0361 STA $0314
.A 0364 LDA $03A1
.A 0367 STA $0315
```

```
.A 036A CLI
.A 036B RTS
```

SYS 836 will invoke the new interrupt code, SYS 861 will turn it off. As with the Commodore 64, there is a possibility of the character printing white-on-white, so that it cannot be seen.

PET/CBM Version

This version is not for original ROM machines, which have the IRQ vector located at address $0219/A:

```
.A 033C LDA $9B
.A 033E STA $8000
.A 0341 JMP ($03A0)
```

To place the link address into $03A0/1:

```
.A 0344 LDA $0090
.A 0347 STA $03A0
.A 034A LDA $0091
.A 034D STA $03A1
```

To fire up the program:

```
.A 0350 SEI
.A 0351 LDA #$3C
.A 0353 STA $0090
.A 0356 LDA #$03
.A 0358 STA $0091
.A 035B CLI
.A 035C RTS
```

To restore the original interrupt:

```
.A 035D SEI
.A 035E LDA $03A0
.A 0361 STA $0090
.A 0364 LDA $03A1
.A 0367 STA $0091
.A 036A CLI
.A 036B RTS
```

SYS 836 will invoke the new interrupt code; SYS 861 will turn it off. Since the PET/CBM does not have colors, the characters will always show.

Project: Adding a Command
PET/CBM Version

It's not possible to write a comparable program to add a command to the PET/CBM. This machine doesn't have a "link" neatly waiting for us at address $0308/9. Equivalent code would need to be somewhat longer and less elegant.

The equivalent program for PET/CBM won't be given here. It would involve writing over part of the CHRGET program (at $0070 to $0087), supplying replacement code for the part we have destroyed, and then adding the new features.

F

Floating
Point
Representation

Packed: 5 bytes (as found in variable or array)

Zero
Flag/
Exponent

Mantissa (value)
4 bytes

High bit represents sign of mantissa
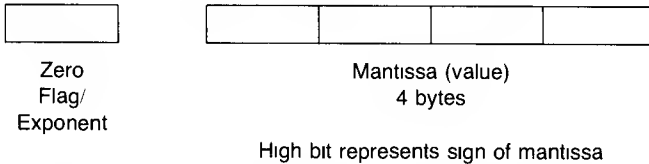
Unpacked: 6 bytes (as found in floating accumulator)

ZF/
Exponent

Mantissa—4 Bytes

Sign
(High Or-
der
Bit only)
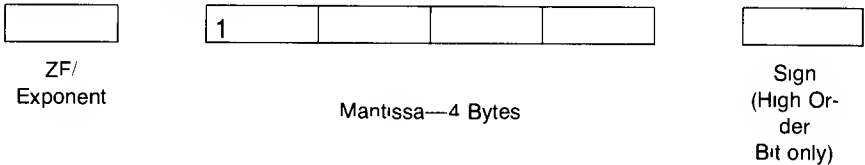
- If exponent = 0, the whole number is zero
- If exponent > $80, the decimal point is to be set as many places to the right as the exponent exceeds $80.
- Example: Exponent: $83 mantissa: 11000000   . binary set the point three positions over: 110.000 . . . to give a value of 6.
- If exponent < = $80, the number is a fraction less than 1.

Exercise: Represent +27 in Floating Point

27 decimal = 11011 binary; mantissa = 11011000 . . . the point is to be positioned 5 places in (11011 000 . . .) so we get:
Exponent: $85 mantissa: 11011000 . . . binary or D8 00 00 00 hexadecimal
To *pack*, we replace the first bit of the mantissa with a sign bit (0 for positive) and arrive at:

85 58 00 00 00

G

Uncrashing

It's best to write a program that doesn't fail (or "crash"). Not all of us succeed in doing this.

If a program gives trouble, it should be tested using breakpoint techniques. The BRK (break) instruction is inserted at several strategic points within the program. The program stops (or "breaks") at these points, and the programmer has an opportunity to confirm correct behavior of the program at selected points. Using this technique, a fault can be pinned down quite closely.

Occasionally, usually because of bad planning, a program crashes and the cause of the crash cannot be identified. Worse still, a lengthy program crashes and the user has forgotten to save a copy of it; the user is then faced with the task of putting it in all over again.

In such cases, *uncrashing* techniques are sometimes available to bring the computer back from limbo. They are never entirely satisfactory, and should be thought of as a last resort.

The technique differs from computer to computer.

PET/CBM

Original ROM PETs cannot be uncrashed.

Subsequent models can be uncrashed, though hardware additions are necessary. The reader should find someone with computer hardware knowledge to assist in fitting the switches to the computer

A toggle switch is needed, to be connected to the "diagnostic sense" line of the parallel user port; that's pin 5 of the PUP. The other side of the toggle switch should connect to ground (pin 12).

Additionally, a momentary pushbutton is required. This must connect the reset line of the computer to ground Technically speaking, it's better to trigger the input of the computer's power-on reset chip (a 555 one-shot), using a resistor to guard against accidentally grounding a live circuit.

To uncrash, set the toggle switch to "on" and press the pushbutton; the machine will come back to life in the machine language monitor Set the toggle switch off. There's more work to do.

The computer is still in an unstable state. To correct this, either of two actions may be taken. You may return to BASIC with .X and immediately give the command CLR; Alternatively, you may type .; followed by the RETURN key.

Whatever investigation or other action is needed should be performed quickly and the computer reset to its normal state.

VIC/Commodore 64

You might try holding down the RUN/STOP key and tapping the RESTORE key to see if that will bring the machine to its senses. Otherwise, you must do a more serious reset.

You must depend on the fact that the computer does a nondestructive memory test during reset There are various commercially available interfaces for the cartridge port—usually "mother boards" that are fitted with reset switches

When the reset switch is pressed, the computer starts from the beginning; but memory is not disturbed. If you have logged the entry location of the machine language monitor, you can bring it back with the appropriate SYS command.

Commodore PLUS/4

There's a reset button next to the power switch. Before you press it, hold down the RUN/STOP and CTRL keys. Now press the reset button and you'll find yourself in the machine language monitor.

H

Supermon Instructions

Program Supermon is not a monitor; it is a monitor *generator* that will make a machine language monitor for you. There's a reason for this. Supermon finds a likely spot in memory and then plunks the MLM there so as to fit it into the most suitable place.

Load Supermon and say RUN. It will write an MLM for you, and call it up Now, exit back to BASIC and command NEW. You do not want the MLM builder any more (it's done the job) and you do not want the danger of building two—or more—MLM's. Get rid of the generator program. Any time you need to use the MLM. give SYS4 or SYS8. as appropriate

Supermon contains the following "essential" commands:

- .R—to display (and change) registers
- .M—to display (and change) memory
- .S—to save memory to disk or tape
- .L—to load from disk or tape
- .G—to go to an ML program
- .X—to exit to BASIC

Supermon also contains the following extra commands:

- .A—to assemble
- .D—to disassemble

237

Most versions of Supermon (not the "do-it-yourself" below) contain the following commands. Though not used by this book, they are useful:

.F—fills memory with fixed contents:
.F 1800 18FF 00
.H—hunts for a memory pattern:
.H 0800 1800 20 D2 FF
.T—transfers a block of memory to a new location:
.T 0800 0BFF 8000

A few versions of Supermon contain the command .I which causes machine language single stepping.

A Do-It-Yourself Supermon

If you do not have access to Supermon from friends, dealers, clubs, or disk, you may find the following program useful for the Commodore 64 only.

Enter this program (it will take you hours) Be sure that lines 300 and above are correct; the lower numbered DATA lines will be checked for accuracy by the program.

When you say RUN, the program will run in two phases. Part 1 takes over two minutes to run: it will check all DATA statements for missing lines and errors and report any problems to you. Part 2 will run only if part 1 shows no errors: it will cause the program to "collapse" into itself, resulting in Supermon. The moment the program has completed running, save Supermon to disk or tape.

The Supermon generated by this program is a "junior" version (to save your fingers) but it contains all commands needed for this book.

```
1  DATA 26,8,100,0,153,34,147,18,29,29,-30
2  DATA 29,29,83,85,80,69,82,32,54,52,-16
3  DATA 45,77,79,78,0,49,8,110,0,153,-39
4  DATA 34,17,32,32,32,32,32,32,32,32,-50
5  DATA 32,32,32,32,32,32,32,0,75,8,-3
6  DATA 120,0,153,34,17,32,46,46,74,73,-48
7  DATA 77,32,66,85,84,84,69,82,70,73,-56
8  DATA 69,76,68,0,102,8,130,0,158,40,-4
9  DATA 194,40,52,51,41,170,50,53,54,172,-53
10  DATA 194,40,52,52,41,170,49,50,55,41,-25
11  DATA 0,0,0,170,170,170,170,170,170,170,-64
12  DATA 170,170,170,170,170,170,170,170,170,170,-29
```

```
13   DATA 170,170,170,170,170,170,170,165,45,133,-61
14   DATA 34,165,46,133,35,165,55,133,36,165,-12
15   DATA 56,133,37,160,0,165,34,208,2,198,-55
16   DATA 35,198,34,177,34,208,60,165,34,208,-34
17   DATA 2,198,35,198,34,177,34,240,33,133,-52
18   DATA 38,165,34,208,2,198,35,198,34,177,-60
19   DATA 34,24,101,36,170,165,38,101,37,72,-56
20   DATA 165,55,208,2,198,56,198,55,104,145,-1
21   DATA 55,138,72,165,55,208,2,198,56,198,-1
22   DATA 55,104,145,55,24,144,182,201,79,208,-48
23   DATA 237,165,55,133,51,165,56,133,52,108,-17
24   DATA 55,0,79,79,79,79,173,230,255,0,-22
25   DATA 141,22,3,173,231,255,0,141,23,3,-64
26   DATA 169,128,32,144,255,0,0,216,104,141,-30
27   DATA 62,2,104,141,61,2,104,141,60,2,-41
28   DATA 104,141,59,2,104,170,104,168,56,138,-17
29   DATA 233,2,141,58,2,152,233,0,0,141,-12
30   DATA 57,2,186,142,63,2,32,147,253,0,-57
31   DATA 162,66,169,42,32,205,251,0,169,82,-62
32   DATA 208,42,230,193,208,6,230,194,208,2,-52
33   DATA 230,38,96,32,207,255,201,13,208,248,-24
34   DATA 104,104,169,0,0,133,38,162,13,169,-11
35   DATA 46,32,205,251,0,32,220,249,0,201,-32
36   DATA 46,240,249,201,32,240,245,162,14,221,-23
37   DATA 195,255,0,208,12,138,10,170,189,207,-36
38   DATA 255,0,72,189,206,255,0,72,96,202,-2
39   DATA 16,236,76,80,252,0,165,193,141,58,-29
40   DATA 2,165,194,141,57,2,96,169,8,133,-44
41   DATA 29,160,0,0,32,143,253,0,177,193,-31
42   DATA 32,190,251,0,32,209,249,0,198,29,-61
43   DATA 208,241,96,32,254,251,0,144,11,162,-53
44   DATA 0,0,129,193,193,193,240,3,76,80,-58
45   DATA 252,0,32,209,249,0,198,29,96,169,-56
46   DATA 59,133,193,169,2,133,194,169,5,96,-20
47   DATA 152,72,32,147,253,0,104,162,46,76,-44
48   DATA 205,251,0,162,0,0,189,234,255,0,-31
49   DATA 32,210,255,232,224,22,208,245,160,59,-51
50   DATA 32,86,250,0,173,57,2,32,190,251,-4
51   DATA 0,173,58,2,32,190,251,0,32,75,-31
52   DATA 250,0,32,33,250,0,240,87,32,220,-13
53   DATA 249,0,32,239,251,0,144,46,32,223,-40
54   DATA 251,0,32,220,249,0,32,239,251,0,-51
55   DATA 144,35,32,223,251,0,32,225,255,240,-33
```

```
56 DATA 60,166,38,208,56,165,195,197,193,165,-22
57 DATA 196,229,194,144,46,160,58,32,86,250,-21
58 DATA 0,32,183,251,0,32,31,250,0,240,-60
59 DATA 224,76,80,252,0,32,239,251,0,144,-42
60 DATA 3,32,20,250,0,32,75,250,0,208,-43
61 DATA 7,32,239,251,0,144,235,169,8,133,-28
62 DATA 29,32,220,249,0,32,53,250,0,208,-18
63 DATA 248,76,229,249,0,32,207,255,201,13,-22
64 DATA 240,12,201,32,208,209,32,239,251,0,-57
65 DATA 144,3,32,20,250,0,174,63,2,154,-46
66 DATA 120,173,57,2,72,173,58,2,72,173,-35
67 DATA 59,2,72,173,60,2,174,61,2,172,-55
68 DATA 62,2,64,174,63,2,154,108,2,160,-56
69 DATA 160,1,132,186,132,185,136,132,183,132,-27
70 DATA 144,132,147,169,64,133,187,169,2,133,-19
71 DATA 188,32,207,255,201,32,240,249,201,13,-42
72 DATA 240,56,201,34,208,20,32,207,255,201,-35
73 DATA 34,240,16,201,13,240,41,145,187,230,-39
74 DATA 183,200,192,16,208,236,76,80,252,0,-18
75 DATA 32,207,255,201,13,240,22,201,44,208,-51
76 DATA 220,32,254,251,0,41,15,240,233,201,-46
77 DATA 3,240,229,133,186,32,207,255,201,13,-45
78 DATA 96,108,48,3,108,50,3,32,22,251,-60
79 DATA 0,208,212,169,0,0,32,111,251,0,-37
80 DATA 165,144,41,16,208,201,76,229,249,0,-22
81 DATA 32,22,251,0,201,44,208,191,32,239,-48
82 DATA 251,0,32,223,251,0,32,207,255,201,-25
83 DATA 44,208,178,32,239,251,0,165,193,133,-7
84 DATA 174,165,194,133,175,32,223,251,0,32,-34
85 DATA 207,255,201,13,208,157,32,114,251,0,-36
86 DATA 76,229,249,0,165,194,32,190,251,0,-39
87 DATA 165,193,72,74,74,74,74,32,214,251,-13
88 DATA 0,170,104,41,15,32,214,251,0,72,-16
89 DATA 138,32,210,255,104,76,210,255,9,48,-9
90 DATA 201,58,144,2,105,6,96,162,2,181,-30
91 DATA 192,72,181,194,149,192,104,149,194,202,-25
92 DATA 208,243,96,32,254,251,0,144,2,133,-30
93 DATA 194,32,254,251,0,144,2,133,193,96,-43
94 DATA 169,0,0,133,42,32,220,249,0,201,-39
95 DATA 32,208,9,32,220,249,0,201,32,208,-25
96 DATA 14,24,96,32,37,252,0,10,10,10,-62
97 DATA 10,133,42,32,220,249,0,32,37,252,-39
98 DATA 0,5,42,56,96,201,58,144,2,105,-26
```

```
99   DATA 8,41,15,96,96,32,220,249,0,201,-62
100  DATA 32,240,249,96,169,0,0,141,0,0,-22
101  DATA 1,32,47,252,0,32,5,252,0,32,-29
102  DATA 242,251,0,144,9,96,32,220,249,0,-28
103  DATA 32,239,251,0,176,222,174,63,2,154,-35
104  DATA 169,63,32,210,255,76,229,249,0,32,-48
105  DATA 143,253,0,202,208,250,96,165,195,164,-12
106  DATA 196,56,233,2,176,1,136,56,229,193,-61
107  DATA 133,30,152,229,194,168,5,30,96,32,-41
108  DATA 55,252,0,133,32,165,194,133,33,162,-22
109  DATA 0,0,134,40,169,147,32,210,255,169,-32
110  DATA 22,133,29,32,165,252,0,32,5,253,-2
111  DATA 0,133,193,132,194,198,29,208,242,169,-16
112  DATA 145,32,210,255,76,229,249,0,160,44,-41
113  DATA 32,86,250,0,32,143,253,0,32,183,-23
114  DATA 251,0,32,143,253,0,162,0,0,161,-25
115  DATA 193,32,20,253,0,72,32,90,253,0,-25
116  DATA 104,32,112,253,0,162,6,224,3,208,-43
117  DATA 18,164,31,240,14,165,42,201,232,177,-10
118  DATA 193,176,28,32,253,252,0,136,208,242,-15
119  DATA 6,42,144,14,189,54,255,0,32,187,-3
120  DATA 253,0,189,60,255,0,240,3,32,187,-24
121  DATA 253,0,202,208,213,96,32,8,253,0,-45
122  DATA 170,232,208,1,200,152,32,253,252,0,-39
123  DATA 138,134,28,32,190,251,0,166,28,96,-47
124  DATA 165,31,56,164,194,170,16,1,136,101,-53
125  DATA 193,144,1,200,96,168,74,144,11,74,-21
126  DATA 176,23,201,34,240,19,41,7,9,128,-63
127  DATA 74,170,189,229,254,0,176,4,74,74,-52
128  DATA 74,74,41,15,208,4,160,128,169,0,-20
129  DATA 0,170,189,41,255,0,133,42,41,3,-62
130  DATA 133,31,152,41,143,170,152,160,3,224,-36
131  DATA 138,240,11,74,144,8,74,74,9,32,-6
132  DATA 136,208,250,200,136,208,242,96,177,193,-29
133  DATA 32,253,252,0,162,1,32,92,252,0,-16
134  DATA 196,31,200,144,241,162,3,192,4,144,-18
135  DATA 242,96,168,185,67,255,0,133,40,185,-13
136  DATA 131,255,0,133,41,169,0,0,160,5,-3
137  DATA 6,41,38,40,42,136,208,248,105,63,-27
138  DATA 32,210,255,202,208,236,169,32,208,11,-16
139  DATA 169,13,36,19,16,5,32,210,255,169,-30
140  DATA 10,76,210,255,32,55,252,0,169,3,-21
141  DATA 133,29,32,220,249,0,32,53,250,0,-42
```

```
142   DATA 208,248,165,32,133,193,165,33,133,194,-43
143   DATA 76,134,252,0,197,40,240,3,32,210,-60
144   DATA 255,96,32,55,252,0,32,223,251,0,-57
145   DATA 142,17,2,162,3,32,47,252,0,72,-43
146   DATA 202,208,249,162,3,104,56,233,63,160,-37
147   DATA 5,74,110,17,2,110,16,2,136,208,-16
148   DATA 246,202,208,237,162,2,32,207,255,201,-31
149   DATA 13,240,30,201,32,240,245,32,220,254,-9
150   DATA 0,176,15,32,18,252,0,164,193,132,-9
151   DATA 194,133,193,169,48,157,16,2,232,157,-47
152   DATA 16,2,232,208,219,134,40,162,0,0,-10
153   DATA 134,38,240,4,230,38,240,117,162,0,-9
154   DATA 0,134,29,165,38,32,20,253,0,166,-48
155   DATA 42,134,41,170,188,67,255,0,189,131,-47
156   DATA 255,0,32,197,254,0,208,227,162,6,-54
157   DATA 224,3,208,25,164,31,240,21,165,42,-63
158   DATA 201,232,169,48,176,33,32,203,254,0,-39
159   DATA 208,204,32,205,254,0,208,199,136,208,-28
160   DATA 235,6,42,144,11,188,60,255,0,189,-15
161   DATA 54,255,0,32,197,254,0,208,181,202,-1
162   DATA 208,209,240,10,32,196,254,0,208,171,-51
163   DATA 32,196,254,0,208,166,165,40,197,29,-15
164   DATA 208,160,32,223,251,0,164,31,240,40,-6
165   DATA 165,41,201,157,208,26,32,99,252,0,-35
166   DATA 144,10,152,208,4,165,30,16,10,76,-40
167   DATA 80,252,0,200,208,250,165,30,16,246,-9
168   DATA 164,31,208,3,185,194,0,0,145,193,-62
169   DATA 136,208,248,165,38,145,193,32,5,253,-41
170   DATA 0,133,193,132,194,160,65,32,86,250,-34
171   DATA 0,32,143,253,0,32,183,251,0,32,-56
172   DATA 143,253,0,76,198,253,0,168,32,203,-29
173   DATA 254,0,208,17,152,240,14,134,28,166,-63
174   DATA 29,221,16,2,8,232,134,29,166,28,-60
175   DATA 40,96,201,48,144,3,201,71,96,56,-30
176   DATA 96,64,2,69,3,208,8,64,9,48,-14
177   DATA 34,69,51,208,8,64,9,64,2,69,-50
178   DATA 51,208,8,64,9,64,2,69,179,208,-47
179   DATA 8,64,9,0,0,34,68,51,208,140,-18
180   DATA 68,0,0,17,34,68,51,208,140,68,-5
181   DATA 154,16,34,68,51,208,8,64,9,16,-20
182   DATA 34,68,51,208,8,64,9,98,19,120,-62
183   DATA 169,0,0,33,129,130,0,0,0,0,-41
184   DATA 89,77,145,146,134,74,133,157,44,41,-39
```

```
194,-43
,-60
-57
3
60,-37
16
201,-31
54,-9
-9
57,-47
10
,-9
48
1,-47
-54
,-63
0,-39
208,-28
15
-1
71,-51
9,-15
0,-6
-35
40
,-9
62
3,-41
,-34

-29
,-63
0
0



-39
```

```
185   DATA 44,35,40,36,89,0,0,88,36,36,-22
186   DATA 0,0,28,138,28,35,93,139,27,161,-10
187   DATA 157,138,29,35,157,139,29,161,0,0,-9
188   DATA 41,25,174,105,168,25,35,36,83,27,-64
189   DATA 35,36,83,25,161,0,0,26,91,91,-24
190   DATA 165,105,36,36,174,174,168,173,41,0,-3
191   DATA 0,124,0,0,21,156,109,156,165,105,-20
192   DATA 41,83,132,19,52,17,165,105,35,160,-26
193   DATA 216,98,90,72,38,98,148,136,84,68,-20
194   DATA 200,84,104,68,232,148,0,0,180,8,-31
195   DATA 132,116,180,40,110,116,244,204,74,114,-32
196   DATA 242,164,138,0,0,170,162,162,116,116,-11
197   DATA 116,114,68,104,178,50,178,0,0,34,-30
198   DATA 0,0,26,26,38,38,114,114,136,200,-27
199   DATA 196,202,38,72,68,68,162,200,58,59,-35
200   DATA 82,77,71,88,76,83,68,44,65,204,-59
201   DATA 250,0,191,250,0,96,250,0,134,250,-25
202   DATA 0,224,250,0,14,251,0,116,251,0,-23
203   DATA 135,251,0,120,252,0,160,253,0,194,-19
204   DATA 253,0,228,249,0,157,249,0,139,249,-63
205   DATA 0,13,32,32,32,80,67,32,32,83,-3
206   DATA 82,32,65,67,32,88,82,32,89,82,-16
207   DATA 32,83,80,-59
255   DATA 208
300   M=63
310   READ X:L=PEEK(M):H=L=255:IF H THEN L=X
320   V=R<>L:S=(T<>63 AND R>0 AND V)
330   IF V THEN T=L:IF NOT S THEN R=R+1:S=R<>L
340   T=(T*3+X)AND63
350   IF S THEN PRINT "ERROR LINE";R:E=-1
360   R=L:IF NOT H GOTO 310
370   IF E THEN STOP
380   PRINT"HERE WE GO":X=-1:RESTORE:B=2049:FOR A=1
      TO 9999
390   IF X>=0 THEN POKE B,X:B=B+1
400   READ X:L=PEEK(M):IF L<255 THEN NEXT A
410   POKE 45,16:POKE 46,16:CLR
```

I

IA Chip Information

The following material has been adapted from manufacturer's specifications. The information is not essential to machine language programming, but can be a great help for further study. Some of these specifications are not widely published and contain "hard to get" information.

6520 PIA, peripheral interface adaptor

6522 VIA, versatile interface adaptor

6525 TPA, tri port adaptor

6526 CIA. complex interface adaptor

6545 CRTC, CRT controller

6560 VIC video interface chip

6566 VIC-2 video interface chip

6581 SID sound interface chip

[Essentially manufacturer's specs, less hardware details]

6520 Peripheral Interface Adaptor (PIA)

The 6520 is an I/O device which acts as an interface between the microprocessor and peripherals such as printers, displays, keyboards, etc. The prime function of the 6520 is to respond to stimulus from each of the two worlds it is serving. On the one side, the 6520 is interfacing with peripherals via two eight-bit bi-directional *peripheral data* ports. On the other side, the device interfaces with the microprocessor through an eight-bit data bus. In addition to the lines described above, the 6520 provides four interrupt input/peripheral control lines and the logic necessary for simple, effective control of peripheral interrupts.
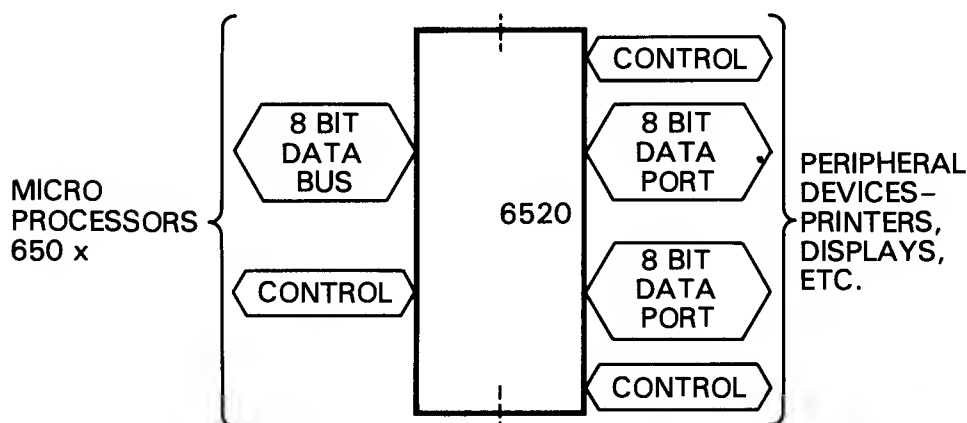
Figure I.1

The functional configuration of the 6520 is programmed by the microprocessor during systems initialization. Each of the peripheral data lines is programmed to act as an input or output and each of the four control/interrupt lines may be programmed for one of four possible control modes. This allows a high degree of flexibility in the overall operation of the interface.

Data Input Register

When the microprocessor writes data into the 6520, the data which appears on the data bus is latched into the Data Input Register. It is then transferred into one of six internal registers of the 6520. This assures that the data on the peripheral output lines will not "glitch," i.e., the output lines will make smooth transitions from high to low or from low to high and the voltage will remain stable except when it is going to the opposite polarity.

Control Registers (CRA and CRB)

Figure I.2 illustrates the bit designation and functions in the Control Registers. The Control Registers allow the microprocessor to control the operation of the interrupt lines (CA1, CA2, CB1, CB2), and peripheral control lines (CA2, CB2). A single bit in each register controls the addressing of the Data Direction Registers (DDRA, DDRB) and the Output Registers, (ORA, ORB) discussed below. In addition, two bits (bit 6 and 7) are provided in each control register to indicate the status of the interrupt input lines (CA1, CA2, CB1, CB2). These interrupt status bits ($\overline{IRQA}$, $\overline{IRQB}$) are normally interrogated by the microprocessor during the interrupt service program to determine the source of an active interrupt. These are the interrupt lines which drive the interrupt input ($\overline{IRQ}$, $\overline{NMI}$) of the microprocessor. The other bits in CRA and CRB are described in the discussion of the interface to the peripheral device.

The various bits in the control registers will be accessed many times during a program to allow the processor to enable or disable interrupts, change operating modes, etc. as required by the peripheral device being controlled.

Data Direction Registers (DDRA, DDRB)

The Data Direction Registers allow the processor to program each line in the 8-bit Peripheral I/O port to act as either an input or an output. Each bit in DDRA controls the corresponding lines in the Peripheral A port and each bit in DDRB controls the corresponding line in the Peripheral B port. Placing a "0" in the Data Direction Register causes the corresponding Peripheral I/O line to act as an input. A "1" causes it to act as an output.

The Data Direction Registers are normally programmed only during the system initialization routine which is performed in response to a Reset signal. However, the contents of these registers can be altered during system operation. This allows very convenient control of some peripheral devices such as keyboards.

Peripheral Output Registers (ORA, ORB)

The Peripheral Output Registers store the output data which appears on the Peripheral I/O port. Writing an "0" into a bit in ORA causes the corresponding line on the Peripheral A port to go low (< 0.4V) if that line is programmed to act as an output. A "1" causes the corresponding output to go high. The lines of the Peripheral B port are controlled by ORB in the same manner.

Interrupt Status Control

The four interrupt/peripheral control lines (CA1, CA2, CB1, CB2) are controlled by the Interrupt Status Control (A, B) This logic interprets the contents of the corresponding Control Register, detects active transitions on the interrupt inputs and performs those operations necessary to assure proper operation of these four peripheral interface lines.

Reset ($\overline{RES}$)

The active low Reset line resets the contents of all 6520 registers to a logic zero. This line can be used as a power-on reset or as a master reset during system operation.

Interrupt Request Line ($\overline{IRQA}$, $\overline{IRQB}$)

The active low Interrupt Request lines ($\overline{IRQA}$ and $\overline{IRQB}$) act to interrupt the microprocessor either directly or through external interrupt priority circuitry.

Each Interrupt Request line has two interrupt flag bits which can cause the Interrupt Request line to go low These flags are bits 6 and 7 in the two Control Registers. These flags act as the link between the peripheral interrupt signals and the microprocessor interrupt inputs. Each flag has a corresponding interrupt disable bit which allows the processor to enable or disable the interrupt from each of the four interrupt inputs (CA1, CA2, CB1, CB2).

The four interrupt flags are set by active transitions of the signal on the interrupt input (CA1, CA2, CB1, CB2). Controlling this active transition is discussed in the next section.

Control of $\overline{IRQA}$

Control Register A bit 7 is always set by an active transition of the CA1 interrupt input signal. Interrupting from this flag can be disabled by setting bit 0 in the Control Register A (CRA) to a logic 0. Likewise, Control Register A bit 6 can be set by an active transition of the CA2 interrupt input signal. Interrupting from this flag can be disabled by setting bit 3 in the Control Register to a logic 0.

Both bit 6 and bit 7 in CRA are reset by a "Read Peripheral Output Register A" operation. This is defined as an operation in which the processor reads the Peripheral A I/O port.

Control of $\overline{\text{IRQB}}$

Control of $\overline{\text{IRQB}}$ is performed in exactly the same manner as that described above for $\overline{\text{IRQA}}$. Bit 7 in CRB is set by an active transition on CB1; interrupting from this flag is controlled by CRB bit 0. Likewise, bit 6 in CRB is set by an active transition on CB2; interrupting from this flag is controlled by CRB bit 3.

Also, both bit 6 and bit 7 are reset by a "Read Peripheral B Output Register" operation.

Summary

IRQA goes low when CRA − 7 = 1 and CRA − 0 = 1 or when CRA − 6 = 1 and CRA − 3 = 1.

IRQB goes low when CRB − 7 = 1 and CRB − 0 = 1 or when CRB − 6 = 1 and CRB − 3 = 1.

It should be stressed at this point that the flags act as the link between the peripheral interrupt signal and the processor interrupt inputs. The interrupt disable bits allow the processor to control the interrupt function.

Peripheral I/O Ports

Each of the Peripheral I/O lines can be programmed to act as an input or an output. This is accomplished by setting a "1" in the corresponding bit in the Data Direction Register for those lines which are to act as outputs. A "0" in a bit of the Data Direction Register causes the corresponding Peripheral I/O lines to act as an input.

Interrupt Input/Peripheral Control Lines (CA1, CA2, CB1, CB2

The four interrupt input/peripheral control lines provide a number of special peripheral control functions. These lines greatly enhance the power of the two general purpose interface ports (PA0−PA7, PB0−PB7).

Peripheral A Interrupt Input/Peripheral Control Lines (CA1, CA2)

CA1 is an interrupt input only. An active transition of the signal on this input will set bit 7 of the Control Register A to a logic 1. The active transition can be programmed by the microprocessor by setting a "0" in bit 1 of

the CRA if the interrupt flag (bit 7 of CRA) is to be set on a negative transition of the CA1 signal or a "1" if it is to be set on a positive transition.

Setting the interrupt flag will interrupt the processor through IRQA if bit 0 of CRA is a 1 as described previously.

CA2 can act as a totally independent interrupt input or as a peripheral control output. As an input (CRA, bit 5 = 0) it acts to set the interrupt flag, bit 6 of CRA, to a logic 1 on the active transition selected by bit 4 of CRA.

These control register bits and interrupt inputs serve the same basic function as that described above for CA1. The input signal sets the interrupt flag which serves as the link between the peripheral device and the processor interrupt structure. The interrupt disable bit allows the processor to exercise control over the system interrupts.

In the Output mode (CRA, bit 5 = 1), CA2 can operate independently to generate a simple pulse each time the microprocessor reads the data on the Peripheral A I/O port. This mode is selected by setting CRA, bit 4 to a "0" and CRA, bit 3 to a "1". This pulse output can be used to control the counters, shift registers, etc. which make sequential data available on the Peripheral input lines.

A second output mode allows CA2 to be used in conjunction with CA1 to "handshake" between the processor and the peripheral device. On the A side, this technique allows positive control of data transfers from the peripheral device into the microprocessor. The CA1 input signals the processor that data is available by interrupting the processor. The processor reads the data and sets CA2 low. This signals the peripheral device that it can make new data available.

The final output mode can be selected by setting bit 4 of CRA to a 1. In this mode, CA2 is a simple peripheral control output which can be set high or low by setting bit 3 of CRA to a 1 or a 0 respectively.

Peripheral B Interrupt Input/Peripheral Control Lines (CB1, CB2)

CB1 operates as an interrupt input only in the same manner as CA1. Bit 7 of CRB is set by the active transition selected by bit 0 of CRB. Likewise, the CB2 input mode operates exactly the same as the CA2 input modes. The CB2 output modes, CRB, bit 5 = 1, differ somewhat from those of CA2. The pulse output occurs when the processor writes data into the Peripheral B Output Register. Also, the "handshaking" operates on data transfers from the processor into the peripheral device.

7	6	5	4	3	2	1	Ø
IRQA1	IRQA2	CA2 CONTROL			DDRA ACCESS	CA1 CONTROL	

CRA

7	6	5	4	3	2	1	Ø
IRQB1	IRQB2	CB2 CONTROL			DDRB ACCESS	CB2 CONTROL	

CRB
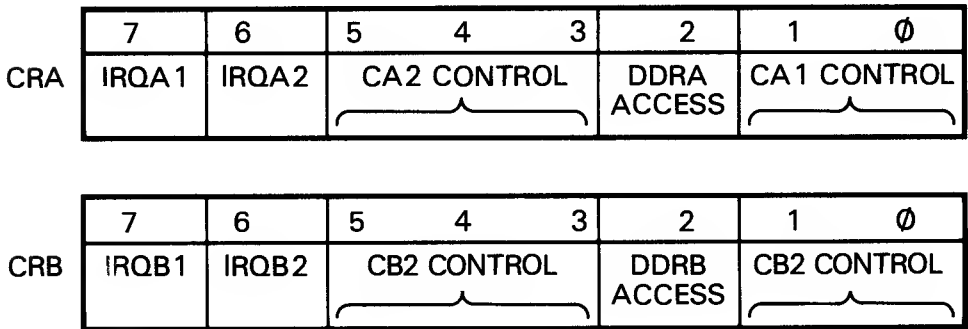
Figure I.2

6545-1 CRT Controller (CRTC)

Concept

The 6545-1 is a CRT Controller intended to provide capability for interfacing the 6500 microprocessor family to CRT or TV-type raster scan displays.

Horizontal Total (R0)

This 8-bit register contains the total of displayed and non-displayed characters, minus one, per horizontal line. The frequency of HSYNC is thus determined by this register.

Horizontal Displayed (R1)

This 8-bit register contains the number of displayed characters per horizontal line.

Horizontal Sync Position (R2)

This 8-bit register contains the position of the HSYNC on the horizontal line, in terms of the character location number on the line. The position of the HSYNC determines the left-to-right location of the displayed text on the video screen. In this way, the side margins are adjusted.

Horizontal and Vertical SYNC Widths (R3)

This 8-bit register contains the widths of both HSYNC and VSYNC, as follows:

7	6	5	4	3	2	1	0
8	4	2	1	8	4	2	1

VSYNC WIDTH*

(NUMBER OF SCAN LINES)

HSYNC WIDTH

(NUMBER OF CHARACTER CLOCK TIMES)
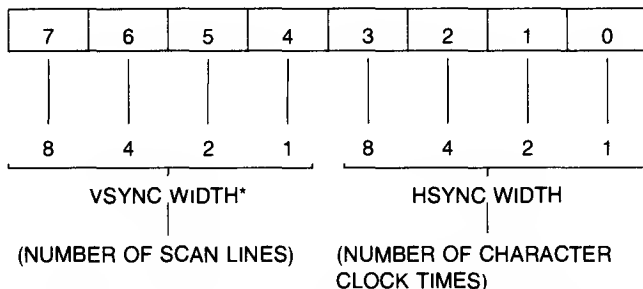
*IF BITS 4-7 ARE ALL "0", THEN VSYNC WILL BE 16 SCAN LINES WIDE

Control of these parameters allows the 6545-1 to be interfaced to a variety of CRT monitors, since the HSYNC and VSYNC timing signals may be accommodated without the use of external one-shot timing.

Vertical Total (R4)

The Vertical Total Register is a 7-bit register containing the total number of character rows in a frame, minus one. This register, along with R5, determines the overall frame rate, which should be close to the line frequency to ensure flicker-free appearance. If the frame time is adjusted to be longer than the period of the line frequency, then RES may be used to provide absolute synchronism.

Vertical Total Adjust (R5)

The Vertical Total Adjust Register is a 5-bit write only register containing the number of additional scan lines needed to complete an entire frame scan and is intended as a fine adjustment for the video frame time.

Vertical Displayed (R6)

This 7-bit register contains the number of displayed character rows in each frame. In this way, the vertical size of the displayed text is determined.

Reg. No.	Register Name	Stored Info	RD	WR	7	6	5	4	3	2	1	0
R0	Horiz. total	# Charac.		✓	■	■	■	■	■	■	■	■
R1	Horiz. Displayed	# Charac.		✓	■	■	■	■	■	■	■	■
R2	Horiz. Sync Position	# Charac.		✓	■	■	■	■	■	■	■	■
R3	VSYNC, HSYNC Widths	# Scan Lines & Char. Times		✓	V_3	V_2	V_1	V_0	H_3	H_2	H_1	H_0
R4	Vert. Total	# Charac. Row		✓	X	■	■	■	■	■	■	■
R5	Vert. Total Adjust.	# Scan Lines		✓	X	X	■	■	■	■	■	■
R6	Vert. Displayed	# Charac. Rows		✓	X	■	■	■	■	■	■	■
R7	Vert. Sync Position	# Charac. Rows		✓	X	■	■	■	■	■	■	■
R8	Mode Control			✓	■	■	■	■	■	■	■	■
R9	Scan Line	# Scan Lines		✓	X	X	X	■	■	■	■	■
R10	Cursor Start	Scan Line No.		✓	X	B_1	B_0	■	■	■	■	■
R11	Cursor End	Scan Line No.		✓	X	X	■	■	■	■	■	■
R12	Display Start Addr (H)			✓	X	X	■	■	■	■	■	■
R13	Display Start Addr (L)			✓	■	■	■	■	■	■	■	■
R14	Cursor Position (H)		✓	✓	X	X	■	■	■	■	■	■
R15	Cursor Position (L)		✓	✓	■	■	■	■	■	■	■	■
R16	Light Pen Reg.(H)		✓		X	X	■	■	■	■	■	■
R17	Light Pen Reg. (L)		✓		■	■	■	■	■	■	■	■

Notes:
■ Designates binary bit
X Designates unused bit. Reading this bit is always "0", except for R31, which does not drive the data bus at all, and for CS "1" which operates likewise.

Vertical Sync Position (R7)

This 7-bit register is used to select the character row time at which the VSYNC pulse is desired to occur and, thus, is used to position the displayed text in the vertical direction.

Mode Control (R8)

This register is used to select the operating modes of the 6545-1 and is outlined as follows:

INTERFACE MODE CONTROL

BIT		OPERATION
1	0	
x	0	NON INTERLACE
x	1	INVALID (DO NOT USE)

VIDEO DISPLAY RAM ADDRESSING
"0" FOR STRAIGHT BINARY
"1" FOR ROW/COLUMN

MUST PROGRAM TO "0"

DISPLAY ENABLE SKEW
"0" FOR NO DELAY
"1" TO DELAY DISPLAY ENABLE
    ONE CHARACTER TIME

CURSOR SKEW
"0" FOR NEW DELAY
"1" TO DELAY CURSOR ONE
    CHARACTER TIME

NOT USED
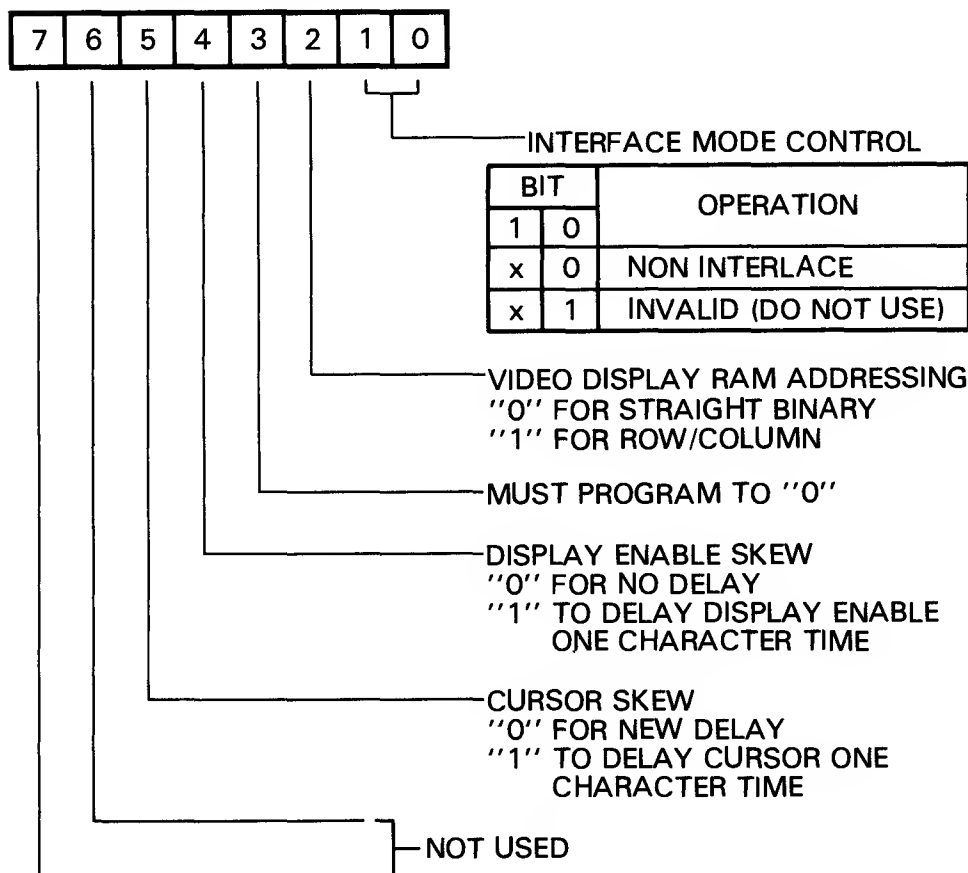
Figure I.3

Scan Line (R9)

This 5-bit register contains the number of scan lines per character row, including spacing.

Cursor Start (R10) and Cursor End (R11)

These 5-bit registers select the starting and ending scan lines for the cursor. In addition, bits 5 and 6 of R10 are used to select the cursor mode, as follows:

BIT		CURSOR MODE
6	5	
0	0	No Blinking
0	1	No Cursor
1	0	Blink at 1/16 field rate
1	1	Blink at 1/32 field rate

Note that the ability to program both the start and end scan line for the cursor enables either block cursor or underline to be accommodated. Registers R14 and R15 are used to control the character position of the cursor over the entire 16K address field.

Display Start Address High (R12) and Low (R13)

These registers together comprise a 14-bit register whose contents is the memory address of the first character of the displayed scan (the character on the top left of the video display, as in Figure 1). Subsequent memory addresses are generated by the 6545-1 as a result of CCLK input pulses. Scrolling of the display is accomplished by changing R12 and R13 to the memory address associated with the first character of the desired line of text to be displayed first. Entire pages of text may be scrolled or changed as well via R12 and R13.

Cursor Position High (R14) and Low (R15)

These registers together comprise a 14-bit register whose contents is the memory address of the current cursor position. When the video display scan counter (MA lines) matches the contents of this register, and when the scan line counter (RA lines) falls within the bounds set by R10 and R11, then the CURSOR output becomes active. Bit 5 of the Mode Control Register (R8) may be used to delay the CURSOR output by a full CCLK time to accommodate slow access memories.

LPEN High (R16) and Low (R17)

These registers together comprise a 14-bit register whose contents is the light pen strobe position, in terms of the video display address at which

the strobe occurred. When the LPEN input changes from low to high, then, on the next negative-going edge of CCLK, the contents of the internal scan counter is stored in registers R16 and R17.

6560 (VIC) Video Interface Chip

The 6560 Video Interface Chip (VIC) is designed for color video graphics applications such as low cost CRT terminals, biomedical monitors, control system displays and arcade or home video games. It provides all of the circuitry necessary for generating color programmable character graphics with high screen resolution. VIC also incorporates sound effects and A/D converters for use in a video game environment.

Features

- Fully expandable system with a 16K byte address space
- System uses industry standard 8 bit wide ROMS and 4 bit wide RAMS
- Mask programmable sync generation, NTSC-6560, PAL-6561
- On-chip color generation (16 colors)
- Up to 600 independently programmable and movable background locations on a standard TV
- Screen centering capability
- Screen grid size up to 192 Horizontal by 200 Vertical dots
- Two selectable graphic character sizes
- On-chip sound system including:
  a) Three independent, programmable tone generators
  b) White noise generator
  c) Amplitude modulator
- Two on-chip 8 bit A/D converters
- ON-chip DMA and address generation
- No CPU wait states or screen hash during screen refresh
- Interlaced/Non-Interlaced switch
- 16 addressable control registers
- Light gun/pen for target games
- 2 modes of color operation

A: Interlace mode: A normal video frame is sent to the TV 60 times each second. Interlace mode cuts the number of repetitions in half. When used with multiplexing equipment, this allows the VIC picture to be blended with a picture from another source.

To turn off: POKE 36864, PEEK(36864) AND 127
To turn on: POKE 36864, PEEK(36864) OR 128

B: Screen origin—horizontal: This determines the positioning of the image on the TV screen. The normal value is 5. Lowering the value moves the screen to the left, and increasing it moves the image to the right.

To change value: POKE 36864, PEEK(36864) AND 128 OR X

LOC	START VALUE-5K VIC		Bit
Hex	Binary	Decimal	Function
9000	00000101	5	ABBBBBBB
9001	00011001	25	CCCCCCCC
9002	10010110	150	HDDDDDDD
9003	v0101110	46 or 176	GEEEEEEF
9004	vvvvvvvv	v	GGGGGGGG
9005	11110000	240	HHHHIIII
9006	00000000	0	JJJJJJJJ
9007	00000000	0	KKKKKKKK
9008	11111111	255	LLLLLLLL
9009	11111111	255	MMMMMMMM
900A	00000000	0	NRRRRRRR
900B	00000000	0	OSSSSSSS
900C	00000000	0	PTTTTTTT
900D	00000000	0	QUUUUUUU
900E	00000000	0	WWWWVVVV
900F	00011011	27	XXXXYZZZ

A: Interlace mode: 0 = off,
   1 = on
B: Screen origin—horizontal
C: Screen origin—vertical
D: Number of video columns
E: Number of video rows
F: Character size:
   0 = 8 × 8, 1 = 8 × 16
G: Raster value
H: Screen memory location
I: Character memory location
J: Light pen—horizontal
K: Light pen—vertical
L: Paddle 1
M: Paddle 2

N: Bass sound switch
O: Alto sound switch
P: Soprano sound switch
Q: Noise switch
R: Bass Frequency
S: Alto Frequency
T: Soprano Frequency
U: Noise Frequency
V: Loudness of sounds
W: Auxiliary color
X: Screen color
Y: Reverse mode 0 = on,
   1 = off
Z: Border color

C: Screen origin—vertical: This determines the up-down placement of the screen image. The normal value is 25. Lowering this causes the screen to move up by 2 rows of dots for each number lowered, and raising it moves the screen down.

To change value: POKE 36865, X

D: Number of video columns: Normally, this is set to 22. Changing this will change the display accordingly. Numbers over 27 will give a 27 column screen. The cursor controls are based on a fixed number of 22 columns, and changing this number makes the cursor controls misbehave.

To change: POKE 36866, PEEK(36866) AND 128 OR X.

E: Number of video rows: The number of rows may range from 0 to 23 A larger number of rows causes garbage to appear on the bottom of the screen.

To change: POKE 36867, PEEK(36867) AND 129 OR (X*2)

F: Character size: This bit determined the size of the matrix used for each character. A 0 here sets normal mode, in which characters are 8 by 8 dots. A 1 sets 8 by 16 mode, where each character is now twice as tall. 8 by 16 mode is normally used for high resolution graphics, where it is likely to have many unique characters on the screen.

To set 8 by 8 mode. POKE 36867, PEEK(36867) AND 254

To set 8 by 16 mode: POKE 36867, PEEK(36867) OR 1

G: Raster value: This number is used to synchronize the light pen with the TV picture.

H: Screen memory location: This determines where in memory the VIC keeps the image of the screen. The highest bit in location 36869 must be a 1. Bits 4–6 of location 36869 are bits 10–12 of the screen's address, and bit 7 of location 36866 is bit 9 of the address of the screen. To determine the location of the screen, use the formula:

S = 4* (PEEK(36866) AND 128) + 64* (PEEK(36869) AND 112)

Note that bit 7 of location 36866 also determines the location of color memory. If this bit is a 0, color memory starts at location 37888. If this bit is a 1, color memory begins at 38400. Here is a formula for this:

C = 37888 + 4* (PEEK(36866) AND 128)

I: Character memory location: This determines where information on the shapes of characters are stored. Normally this pointer is to the character generator ROM, which contains both the upper case/graphics or the

upper/lower case set. However, a simple POKE command can change this pointer to a RAM location, allowing custom character sets and high resolution graphics.

To change· POKE 36869, PEEK(36869) AND 240 OR X
(See chart on next page.)

J: Light pen horizontal: This contains the latched number of the dot under the light pen, from the left of the screen.

K: Light pen vertical: The latched number of the dot under the pen, counted from the top of the screen.

X Value	Location		Contents
	HEX	**Decimal**	
0	8000	32768	Upper case normal characters
1	8400	33792	Upper case reversed characters
2	8800	34816	Lower case normal characters
3	8C00	35840	Lower case reversed characters
4	9000	36864	unavailable
5	9400	37888	unavailable
6	9800	38912	VIC chip-unavailable
7	9C00	39936	ROM-unavailable
8	0000	0	unavailable
9	———	———	unavailable
10	———	———	unavailable
11	———	———	unavailable
12	1000	4096	RAM
13	1400	5120	RAM
14	1800	6144	RAM
15	1C00	7168	RAM

L: Paddle X: This contains the digitized value of a variable resistance (game paddle). The number reads from 0 to 255.

M: Paddle Y: Same as Paddle X, for a second analog input.

N: Bass switch: If this bit is a 0, no sound is played from Voice 1. A 1 in this bit results in a tone determined by Frequency 1.

To turn on: POKE 36874, PEEK(36874) OR 128

To turn off: POKE 36874, PEEK(36874) AND 127

O: Alto switch: See Bass switch.

P: Soprano switch: See Bass switch.

Q: Noise switch: See Bass switch.

R: Bass Frequency: This is a value corresponding to the frequency of the tone being played. The larger the number, the higher the pitch of the tone.

The actual frequency of the sound in cycles per second (hertz) is determined by the following formula:

$$\text{Frequency} = \frac{\text{Clock}}{(127 - X)}$$

X is the number from 0 to 127 that is put into the frequency register. If X is 127, then use −1 for X in the formula. The value of Clock comes from the following table:

Register	NTSC (US TV's)	PAL (European)
36874	3995	4329
36875	7990	8659
36876	15980	17320
36877	31960	34640

To set: POKE 36874, PEEK(36874) AND 128 OR X

S: Alto Frequency: This is a value corresponding to the frequency of the tone being played. The larger the number, the higher the pitch of the tone.

T: Soprano Frequency: This is a value corresponding to the frequency of the tone being played. The larger the number, the higher the pitch of the tone.

To set: POKE 36876, PEEK(36876) AND 128 OR X

U: Noise Frequency: This is a value corresponding to the frequency of the noise being played. The larger the number, the higher the pitch of the noise.

To set: POKE 36877, PEEK(36877) AND 128 OR X

V: Loudness of sounds: This is the volume control for all the sounds playing. 0 is off and 15 is the loudest sound.

To set: POKE 36878, PEEK(36878) AND 240 OR X

W: Auxiliary color: This register holds the color number of the auxiliary color. The value can be from 0 to 15.

To set: POKE 36878, PEEK(36878) AND 15 OR (16*X)

X: Screen color: A number from 0 to 15 sets the color of the screen.

To set: POKE 36879, PEEK(36879) AND 240 OR X

Y: Reverse mode: A 1 in this bit indicates normal characters, and a 0 here causes all characters to be displayed as if reversed.

To turn on reverse mode: POKE 36879, PEEK(36879) AND 247

To turn off reverse mode. POKE 36879, PEEK(36879) OR 8

Z: Border color: A number from 0 to 7 sets the color of the screen.

To set: POKE 36879, PEEK(36879) AND 248 OR X

6522 Versatile Interface Adapter (VIA)

The 6522 Versatile Interface Adapter (VIA) provides two peripheral ports with input latching, two powerful interval timers, and a serial-to-parallel/parallel-to-serial shift register.

6522 Versatile Interface Adapter Description

ADDRESS	DESCRIPTION	REGISTER
9110	Port B	AAAAAAAA
9111	Port A (with handshaking)	BBBBBBBB
9112	Data Direction B	CCCCCCCC
9113	Data Direction A	DDDDDDDD
9114	Timer #1, low byte	EEEEEEEE
9115	Timer #1, high byte	FFFFFFFF
9116	Timer #1, low byte to load	GGGGGGGG
9117	Timer #1, high byte to load	HHHHHHHH
9118	Timer #2, low byte	IIIIIIII
9119	Timer #2, high byte	JJJJJJJJ
911A	Shift Register	KKKKKKKK
911B	Auxiliary Control	LLMNNNOP
911C	Peripheral Control	QQQRSSST
911D	Interrupt Flags	UVWXYZab
911E	Interrupt Enable	cedfghij
911F	Port A (no handshaking)	kkkkkkkk

PORT A I/O REGISTER

These· eight bits are connected to the eight pins which make up port B. Each pin can be set for either input or output.

Input latching is available on this port. When latch mode is enabled the data in the register freezes when the CB1 interrupt flag is set. The register stays latched until the interrupt flag is cleared.

Handshaking is available for output from this port. CB2 will act as a DATA READY SIGNAL. This must be controlled by the user program. CB1 acts as the DATA ACCEPTED signal, and must be controlled by the device connected to the port. When DATA ACCEPTED is sent to the 6522, the DATA READY line is cleared, and the interrupt flag is set.

PORT B I/O REGISTER

These eight bits are connected to the eight pins which make up port A. Each pin can be set for either input or output. Handshaking is available for both read and write operations. Write handshaking is similar to that on PORT B. Read handshaking is automatic. The CA1 input pin acts as a DATA READY signal. The CA2 pin (used for output) is used for a DATA ACCEPTED signal. When a DATA READY signal is received a flag is set. The chip can be set to generate an interrupt or the flag can be polled under program control. The DATA ACCEPTED signal can either be a pulse or a DC level. It is set low by the CPU and cleared by the DATA READY signal.

DATA DIRECTION FOR PORT B

This register is used to control whether a particular bit in PORT B is used for input or output. Each bit of the data direction register (DDR) is associated with a bit of port B. If a bit in the DDR is set to 1, the corresponding bit of the port will be an OUTPUT. If a bit in the DDR is 0, the corresponding bit of the port will be an INPUT.

For example, if the DDR is set to 7, port B will be set up as follows:

BITS NUMBER	DDR	PORT B FUNCTION
0	1	OUTPUT
1	1	OUTPUT
2	1	OUTPUT
3	0	INPUT
4	0	INPUT
5	0	INPUT
6	0	INPUT
7	0	INPUT

DATA DIRECTION REGISTER FOR PORT A

This is similar to the DDR for port B, except that it works on PORT A.

E,F,G,H: TIMER CONTROLS

There are two timers on the 6522 chip. The timers can be set to count down automatically or count pulses received by the VIA. The mode of operation is selected by the Auxiliary Control register.

TIMER T1 on the 6522 consists of two 8-bit latches and a 16-bit counter. The various modes of the TIMER are selected by setting the AUXILIARY CONTROL REGISTER (ACR). The latches are used to store a 16-bit data word to load into the counter. Loading a number into the latches does not affect the count in progress

After it is set, the counter will begin decrementing at 1 MHz. When the counter reaches zero, an interrupt flag will be set, and the IRQ will go low. Depending on how the TIMER is set, either further interrupts will be disabled, or it will automatically load the two latches into the counter and continue counting. The TIMER can also be set to invert the output signal on a peripheral pin each time it reaches zero and resets.

The TIMER locations work differently on reading and writing.

WRITING TO THE TIMER:

E: Write into the low order latch. This latch can be loaded into the low byte of the 16-bit counter.

F: Write into the high order latch, write into the high order counter, transfer low order latch into the low order counter, and reset the TIMER T1 interrupt flag. In other words, when this location is set the counter is loaded.

G: Same as E.

H: Write into the high order latch and reset the TIMER T1 interrupt flag.

READ TIMER T1

E: Read the TIMER T1 low order counter and reset the TIMER T1 interrupt flag.

F: Read the TIMER T1 high order counter.

G: Read the TIMER T1 low order latch.

H: Read the TIMER T1 high order latch.

TIMER T2

This TIMER operates as an interval timer (in one-shot mode), or as a counter for counting negative pulses on PORT B pin 6. A bit in the ACR selects which mode TIMER T2 is in.

WRITING TO TIMER T2

I: Write TIMER T2 low order byte of latch.
J: Write TIMER T2 high order counter byte, transfer low order latch to low order counter, clear TIMER T2 interrupt flag.

READING TIMER T2

I: Read TIMER T2 low order counter byte, and clear TIMER T2 interrupt flag.

K: SHIFT REGISTER

A shift register is a register which will rotate itself through the CB2 pin. The shift register can be loaded with any 8-bit pattern which can be shifted out through the CB1 pin, or input to the CB1 pin can be shifted into the shift register and then read This makes it highly useful for serial to parallel and parallel to serial conversions.

The shift register is controlled by bits 2–4 of the Auxiliary Control register.

L,M,N,O,P: AUXILIARY CONTROL REGISTER

L: TIMER 1 CONTROL

```
BIT #   7   6
        0   0   One-shot mode (output to PB7 disabled)
        0   1   Free running mode (output to PB7 disabled)
        1   0   One-shot mode (output to PB7 enabled)
        1   1   Free running mode (output to PB7 enabled)
```

M: TIMER 2 CONTROL

TIMER 2 has 2 modes. If this bit is 0, TIMER 2 acts as an interval timer in one-shot mode. If this bit is 1, TIMER 2 will count a predetermined number of pulses on pin PB6.

N: SHIFT REGISTER CONTROL

BIT #   4 3 2

          0 0 0   SHIFT REGISTER DISABLED

          0 0 1   SHIFT IN (FROM CB1) UNDER CONTROL OF
                  TIMER 2

          0 1 0   SHIFT IN UNDER CONTROL OF SYSTEM CLOCK
                  PULSES

          0 1 1   SHIFT IN UNDER CONTROL OF EXTERNAL
                  CLOCK PULSES

          1 0 0   FREE RUN MODE AT RATE SET BY TIMER 2

          1 0 1   SHIFT OUT UNDER CONTROL OF TIMER 2

          1 1 0   SHIFT OUT UNDER CONTROL OF SYSTEM
                  CLOCK PULSES

          1 1 1   SHIFT OUT UNDER CONTROL OF EXTERNAL
                  CLOCK PULSES

O: PORT B LATCH ENABLE

As long as this bit is 0, the PORT B register will directly reflect the data on the pins.

If this bit is set to one, the data present on the input pins of PORT A will be latched within the chip when the CB1 INTERRUPT FLAG is set. As long as the CB1 INTERRUPT FLAG is set, the data on the pins can change without affecting the contents of the PORT B register. Note that the CPU always reads the register (the latches) rather than the pins

Input latching can be used with any of the input or output modes available for CB2.

P: PORT A LATCH ENABLE

As long as this bit is 0, the PORT A register will directly reflect the data on the pins.

If this bit is set to one, the data present on the input pins of PORT A will be latched within the chip when the CA1 INTERRUPT FLAG is set. As long as the CA1 INTERRUPT FLAG is set, the data on the pins can change without affecting the contents of the PORT A register. Note that the CPU always reads the register (the latches) rather than the pins.

Input latching can be used with any of the input or output modes available for CA2.

Q,R,S,T THE PERIPHERAL CONTROL REGISTER

Q: CB2 CONTROL

BIT #	Q 7	Q 6	Q 5	DESCRIPTION
	0	0	0	Interrupt Input Mode
	0	0	1	Independent Interrupt Input Mode
	0	1	0	Input Mode
	0	1	1	Independent Input Mode
	1	0	0	Handshake Output Mode
	1	0	1	Pulse Output Mode
	1	1	0	Manual Output Mode (CB2 is held LOW)
	1	1	1	Manual Output Mode (CB2 is held HIGH)

INTERRUPT INPUT MODE:

The CB2 interrupt flag (IFR bit 3) will be set on a negative (high-to-low) transition on the CB2 input line. The CB2 interrupt bit will be cleared on a read or write to PORT B.

INDEPENDENT INTERRUPT INPUT MODE:

As above, the CB2 interrupt flag will be set on a negative transition on the CB2 input line. However, reading or writing to PORT B does not clear the flag.

INPUT MODE:

The CB2 interrupt flag (IFR bit 3) will be set on a positive (low-to-high) transition of the CB2 line. The CB2 flag will be cleared on a read or write of PORT B.

INDEPENDENT INPUT MODE:

As above, the CB2 interrupt flag will be set on a positive transition on the CB2 line. However, reading or writing PORT B does not affect the flag.

HANDSHAKE OUTPUT MODE:

The CB2 line will be set low on a write to PORT B. It will be reset high again when there is an active transition on the CB1 line.

PULSE OUTPUT MODE:

The CB2 line is set low for one cycle after a write to PORT B.

MANUAL OUTPUT MODE:

The CB2 line is held low.

MANUAL OUTPUT MODE:

The CB2 line is held high.

R: CB1 CONTROL

This bit selects the active transition of the input signal applied to the CB1 pin. If this bit is 0, the CB1 interrupt flag will be set on a negative transition (high-to-low). If this bit is a 1, the CB 1 interrupt flag will be set on a positive (low-to-high) transition.

S:CA2 CONTROL

	S	S	S	
BIT #	3	2	1	DESCRIPTION
	0	0	0	Interrupt Input Mode
	0	0	1	Independent Interrupt Input Mode
	0	1	0	Input Mode
	0	1	1	Independent Input Mode
	1	0	0	Handshake Output Mode
	1	0	1	Pulse Output Mode
	1	1	0	Manual Output Mode (CA2 is held LOW)
	1	1	1	Manual Output Mode (CA2 is held HIGH)

INTERRUPT INPUT MODE:

The CA2 interrupt flag (IFR bit 0) will be set on a negative (high-to-low) transition on the CA2 input line. The CA2 interrupt bit will be cleared on a read or write to PORT A.

INDEPENDENT INTERRUPT INPUT MODE:

As above, the CA2 interrupt flag will be set on a negative transition on the CA2 input line. However, reading or writing to PORT A does not clear the flag.

INPUT MODE:

The CA2 interrupt flag (IFR bit 0) will be set on a positive (low-to-high) transition of the CA2 line. The CA2 flag will be cleared on a read or write of PORT A.

INDEPENDENT INPUT MODE:

As above, the CA2 interrupt flag will be set on a positive transition on the CA2 line. However, reading or writing PORT A does not affect the flag.

HANDSHAKE OUTPUT MODE:

The CA2 line will be set low on a read or write to PORT A. It will be reset high again when there is an active transition on the CA1 line.

PULSE OUTPUT MODE:

The CA2 line is set low for one cycle after a read or write to PORT A.

MANUAL OUTPUT MODE:

The CA2 line is held low.

MANUAL OUTPUT MODE:

The CA2 line is held high.

T: CA1 CONTROL

This bit of the PCR selects the active transition of the input signal applied to the CA1 input pin. If this bit is 0, the CA1 interrupt flag (Bit) will be set by a negative transition (high-to-low) on the CA1 pin. If this bit is 1, the CA1 interrupt flag will be set by a positive transition (low-to-high).

There are two registers associated with interrupts: The INTERRUPT FLAG REGISTER (IFR) and the INTERRUPT ENABLE REGISTER (IER). The IFR has eight bits, each one connected to a register in the 6522. Each bit in the IFR has an associated bit in the IER. The flag is set when a register wants to interrupt. However, no interrupt will take place unless the corresponding bit in the IER is set.

UVWXYZab: INTERRUPT FLAG REGISTER

When the flag is set, the pin associated with that flag is attempting to interrupt the 6502. Bit U is not a normal flag. It goes high if both the flag and the corresponding bit in the INTERRUPT ENABLE REGISTER are set. It can be cleared only by clearing all the flags in the IFR or disabling all active interrupts in the IER.

	SET BY	**CLEARED BY**
U	IRQ STATUS	
V̲	TIMER 1 time-out	Reading TIMER 1 low order counter and writing TIMER 1 high order latch
W	TIMER 2 time-out	Reading TIMER 2 low order counter and writing TIMER 2 high order counter
X	CB1 pin active transition	Reading or writing PORT B
Y	CB2 pin active transition	Reading or writing PORT B
Z	Completion of 8 shifts	Reading or writing the shift register
a	CA1 pin active transition	Reading or writing PORT A (BBBBBBBB in above chart)
b	CA2 pin active transition	Reading or writing PORT A (BBBBBBBB in above chart)

cdefghij: INTERRUPT ENABLE REGISTER

c: ENABLE CONTROL

If this bit is a 0 during a write to this register, each 1 in bits 0-6 clears the corresponding bit in the IER. If this bit is a 1 during this register, each 1 in bits 0-6 will set the corresponding IER bit.

d   TIMER 1 time-out enable
e   TIMER 2 time-out enable
f   CB1 interrupt enable
g   CB2 interrupt enable
h   Shift interrupt enable
i   CA1 interrupt enable
j   CA2 interrupt enable

PORT A

This is similar to BBBBBBBB, except that the handshaking lines (CA1 and CA2) are unaffected by operations on this port.

6526 (CIA) Complex Interface Adaptor

REGISTER MAP

RS3	RS2	RS1	RS0	REG	NAME	
0	0	0	0	0	PRA	PERIPHERAL DATA REG A
0	0	0	1	1	PRB	PERIPHERAL DATA REG B
0	0	1	0	2	DDRA	DATA DIRECTION REG A
0	0	1	1	3	DDRB	DATA DIRECTION REG B
0	1	0	0	4	TA LO	TIMER A LOW REGISTER
0	1	0	1	5	TA HI	TIMER A HIGH REGISTER
0	1	1	0	6	TB LO	TIMER B LOW REGISTER
0	1	1	1	7	TB HI	TIMER B HIGH REGISTER
1	0	0	0	8	TOD 10ths	10ths OF SECONDS REGISTER
1	0	0	1	9	TOD SEC	SECONDS REGISTER
1	0	1	0	A	TOD MIN	MINUTES REGISTER
1	0	1	1	B	TOD HR	HOURS—AM/PM REGISTER
1	1	0	0	C	SDR	SERIAL DATA REGISTER
1	1	0	1	D	ICR	INTERRUPT CONTROL REGISTER
1	1	1	0	E	CRA	CONTROL REG A
1	1	1	1	F	CRB	CONTROL REG B

I/O Ports (PRA, PRB, DDRA, DDRB)

Ports A and B each consist of an 8-bit Peripheral Data Register (PR) and an 8-bit Data Direction Register (DDR). If a bit in the DDR is set to a one, the corresponding bit in the PR is an output; if a DDR bit is set to a zero, the corresponding PR bit is defined as an input. On a READ, the PR reflects the information present on the actual port pins (PA0–PA7, PB0–PB7) for both input and output bits. Port A and Port B have passive pull-up devices as well as active pull-ups, providing both CMOS and TTL compatibility. Both ports have two TTL load drive capability. In addition to normal I/O operation, PB6 and PB7 also provide timer output functions.

Handshaking

Handshaking on data transfers can be accomplished using the $\overline{PC}$ output pin and the FLAG input pin. $\overline{PC}$ will go low for one cycle following a read or write of PORT B. This signal can be used to indicate "data ready" at PORT B or "data accepted" from PORT B. Handshaking on 16-bit data transfers (using both PORT A and PORT B) is possible by always reading

or writing PORT A first. $\overline{\text{FLAG}}$ is a negative edge sensitive input which can be used for receiving the $\overline{\text{PC}}$ output from another 6526, or as a general purpose interrupt input. Any negative transition of $\overline{\text{FLAG}}$ will set the FLAG interrupt bit.

REG	NAME	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	PRA	PA_7	PA_6	PA_5	PA_4	PA_3	PA_2	PA_1	PA_0
1	PRB	PB_7	PB_6	PB_5	PB_4	PB_3	PB_2	PB_1	PB_0
2	DDRA	DPA_7	DPA_6	DPA_5	DPA_4	DPA_3	DPA_2	DPA_1	DPA_0
3	DDRB	DPB_7	DPB_6	DPB_5	DPB_4	DPB_3	DPB_2	DPB_1	DPB_0

Interval Timers (Timer A, Timer B)

Each interval timer consists of a 16-bit read-only Timer Counter and a 16-bit write-only Timer Latch. Data written to the timer are latched in the Timer Latch, while data read from the timer are the present contents of the Time Counter. The timers can be used independently or linked for extended operations. The various timer modes allow generation of long time delays, variable width pulses, pulse trains and variable frequency waveforms. Utilizing the CNT input, the timers can count external pulses or measure frequency, pulse width and delay times of external signals. Each timer has an associated control register, providing independent control of the following functions:

Start/Stop

A control bit allows the time to be started or stopped by the microprocessor at any time.

PB On/Off:

A control bit allows the timer output to appear on a PORT B output line (PB6 for TIMER A and PB7 for TIMER B) This function overrides the DDRB control bit and forces the appropriate PB line to an output.

Toggle/Pulse

A control bit selects the output applied to PORT B On every timer underflow the output can either toggle or generate a single positive pulse of one cycle duration. The Togq'e output is set high whenever the timer is started and is set low by $\overline{\text{RES}}$.

One-Shot/Continuous

A control bit selects either timer mode. In one-shot mode, the timer will count down from the latched value to zero, generate an interrupt, reload the latched value, then stop. In continuous mode, the timer will count from the latched value to zero, generate an interrupt, reload the latched value and repeat the procedure continuously.

Force Load

A strobe bit allows the timer latch to be loaded into the timer counter at any time, whether the timer is running or not.

Input Mode:

Control bits allow selection of the clock used to decrement the timer. TIMER A can count $\phi2$ clock pulses or external pulses applied to the CNT pin. TIMER B can count $\phi2$ pulses, external CNT pulses, TIMER A underflow pulses or TIMER A underflow pulses while the CNT pin is held high.

The timer latch is loaded into the timer on any timer underflow, on a force load or following a write to the high byte of the prescaler while the timer is stopped. If the timer is running, a write to the high byte will load the timer latch, but not reload the counter.

READ (TIMER)

REG	NAME								
4	TA LO	TAL_7	TAL_6	TAL_5	TAL_4	TAL_3	TAL_2	TAL_1	TAL_0
5	TA HI	TAH_7	TAH_6	TAH_5	TAH_4	TAH_3	TAH_2	TAH_1	TAH_0
6	TB LO	TBL_7	TBL_6	TBL_5	TBL_4	TBL_3	TBL_2	TBL_1	TBL_0
7	TB HI	TBH_7	TBH_6	TBH_5	TBH_4	TBH_3	TBH_2	TBH_1	TBH_0

WRITE (PRESCALER)

REG	NAME								
4	TA LO	PAL_7	PAL_6	PAL_5	PAL_4	PAL_3	PAL_2	PAL_1	PAL_0
5	TA HI	PAH_7	PAH_6	PAH_5	PAH_4	PAH_3	PAH_2	PAH_1	PAH_0
6	TB LO	PB_7	PB_6	PB_5	PB_4	PB_3	PB_2	PB_1	PB_0
7	TB HI	PBH_7	PBH_6	PBH_5	PBH_4	PBH_3	PBH_2	PBH_1	PBH_0

Time of Day Clock (TOD)

The TOD clock is a special purpose timer for real-time applications. TOD consists of a 24-hour (AM/PM) clock with 1/10th second resolution. It is organized into 4 registers: 10ths of seconds, Seconds, Minutes and Hours. The AM/PM flag is in the MSB of the Hours register for easy bit testing. Each register reads out in BCD format to simplify conversion for driving displays, etc. The clock requires an external 60 Hz or 50 Hz (programmable) TTL level input on the TOD pin for accurate timekeeping. In addition to time-keeping, a programmable ALARM is provided for generating an interrupt at a desired time. The ALARM registers are located at the same addresses as the corresponding TOD registers. Access to the ALARM is governed by a Control Register bit. The ALARM is write-only; any read of a TOD address will read time regardless of the state of the ALARM access bit.

A specific sequence of events must be followed for proper setting and reading of TOD. TOD is automatically stopped whenever a write to the Hours register occurs. The clock will not start again until after a write to the 10ths of seconds register. This assures TOD will always start at the desired time. Since a carry from one stage to the next can occur at any time with respect to a read operation, a latching function is included to keep all Time Of Day information constant during a read sequence. All four TOD registers latch on a read of Hours and remain latched until after a read of 10ths of seconds. The TOD clock continues to count when the output registers are latched. If only one register is to be read, there is no carry problem and the register can be read "on the fly," provided that any read of Hours is followed by a read of 10ths of seconds to disable the latching.

READ

REG	NAME								
8	TOD 10THS	0	0	0	0	T_8	T_4	T_2	T_1
9	TOD SEC	0	SH_4	SH_2	SH_1	SL_8	SL_4	SL_2	SL_1
A	TOD MIN	0	MH_4	MH_2	MH_1	ML_8	ML_4	ML_2	ML_1
B	TOD HR	PM	0	0	HH	HL_8	HL_4	HL_2	HL_1

WRITE

$CRB_7 = 0$ TOD
$CRB_7 = 1$ ALARM
(SAME FORMAT AS READ)

Serial Port (SDR)

The serial port is a buffered, 8-bit synchronous shift register system. A control bit selects input or output mode. In input mode, data on the SP pin is shifted into the shift register on the rising edge of the signal applied to the CNT pin After 8 CNT pulses, the data in the shift register is dumped into the Serial Data Register and an interrupt is generated. In the output mode, TIMER A is used for the baud rate generator. Data is shifted out on the SP pin at ½ the underflow rate of TIMER A The maximum baud rate possible is $\phi2$ divided by 4, but the maximum useable baud rate will be determined by line loading and the speed at which the receiver responds to input data. Transmission will start following a write to the Serial Data Register (provided TIMER A is running and in continuous mode). The clock signal derived from TIMER A appears as an output on the CNT pin. The data in the Serial Data Register will be loaded into the shift register then shift out to the SP pin when a CNT pulse occurs. Data shifted out becomes valid on the falling edge of CNT and remains valid until the next falling edge. After 8 CNT pulses, an interrupt is generated to indicate more data can be sent. If the Serial Data Register was loaded with new information prior to this interrupt, the new data will automatically be loaded into the shift register and transmission will continue. If the microprocessor stays one byte ahead of the shift register, transmission will be continuous. If no further data is to be transmitted, after the 8th CNT pulse, CNT will return high and SP will remain at the level of the last data bit transmitted. SDR data is shifted out MSB first and serial input data should also appear in this format.

The bidirectional capability of the Serial Port and CNT clock allows many 6526 devices to be connected to a common serial communication bus on which one 6526 acts as a master, sourcing data and shift clock, while all other 6526 chips act as slaves. Both CNT and SP outputs are open drain to allow such a common bus. Protocol for master/slave selection can be transmitted over the serial bus, or via dedicated handshaking lines.

REG	NAME								
C	SDR	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0

Interrupt Control (ICR)

There are five sources of interrupts on the 6526: underflow from TIMER A, underflow from TIMER B, TOD ALARM, Serial Port full/empty and $\overline{\text{FLAG}}$. A single register provides masking and interrupt information. The interrupt Control Register consists of a write-only MASK register and a read-only DATA register. Any interrupt will set the corresponding bit in the DATA register Any interrupt which is enabled by the MASK register will set the IR bit (MSB) of the DATA register and bring the $\overline{\text{IRQ}}$ pin low. In a multi-chip system, the IR bit can be polled to detect which chip has generated an interrupt request The interrupt DATA register is cleared and the $\overline{\text{IRQ}}$ line returns high following a read of the DATA register. Since each interrupt sets an interrupt bit regardless of the MASK, and each interrupt bit can be selectively masked to prevent the generation of a processor interrupt, it is possible to intermix polled interrupts with true interrupts. However, polling the IR bit will cause the DATA register to clear, therefore, it is up to the user to preserve the information contained in the DATA register if any polled interrupts were present.

The MASK register provides convenient control of individual mask bits. When writing to the MASK register, if bit 7 (SET/$\overline{\text{CLEAR}}$) of the data written is a ZERO, any mask bit written with a one will be cleared, while those mask bits written with a zero will be unaffected If bit 7 of the data written is a ONE, any mask bit written with a one will be set, while those mask bits written with a zero will be unaffected. In order for an interrupt flag to set IR and generate an Interrupt Request, the corresponding MASK bit must be set.

READ (INT DATA)

REG	NAME								
D	ICR	IR	0	0	FLG	SP	ALRM	TB	TA

WRITE (INT MASK)

REG	NAME								
D	ICR	S/$\overline{\text{C}}$	X	X	FLG	SP	ALRM	TB	TA

Control Registers

There are two control registers in the 6526, CRA and CRB. CRA is associated with TIMER A and CRB is associated with TIMER B. The register format is as follows:

CRA:

Bit	Name	Function
0	START	1 = START TIMER A, 0 = STOP TIMER A. This bit is automatically reset when underflow occurs during one-shot mode.
1	PBON	1 = TIMER A output appears on PB6, 0 = PB6 normal operation.
2	OUTMODE	1 = TOGGLE, 0 = PULSE
3	RUNMODE	1 = ONE-SHOT, 0 = CONTINUOUS
4	LOAD	1 = FORCE LOAD (this is STROBE input, there is no data storage, bit 4 will always read back a zero and writing a zero has no effect).
5	INMODE	1 = TIMER A counts positive CNT transitions, 0 = TIMER A counts φ2 pulses.
6	SPMODE	1 = SERIAL PORT output (CNT sources shift clock), 0 = SERIAL PORT input (external shift clock required).
7	TODIN	1 = 50 Hz clock required on TOD pin for accurate time, 0 = 60 Hz clock required on TOD pin for accurate time.

(Bits CRB0-CRB4 are identical to CRA0-CRA4 for TIMER B with the exception that bit 1 controls the output of TIMER B on PB7).

Bit	Name	Function
5,6	INMODE	Bits CRB5 and CRB6 select one of four input modes for TIMER B as:

CRB6	CRB5	
0	0	TIMER B counts φ2 pulses.
0	1	TIMER B counts positive CNT transitions.
1	0	TIMER B counts TIMER A underflow pulses.
1	1	TIMER B counts TIMER A underflow pulses while CNT is high.

Bit	Name	Function
7	ALARM	1 = writing to TOD registers sets ALARM, 0 = writing to TOD registers sets TOD clock.

REG	NAME	TOD IN	SP MODE	IN MODE	LOAD	RUN MODE	OUT MODE	PB ON	START
E	CRA	0 = 60Hz	0 = INPUT	0 = φ2	1 = FORCE LOAD (STROBE)	0 = CONT	0 = PULSE	0 = PB$_6$ OFF	0 = STOP
		1 = 50Hz	1 OUT-PUT	1 = CNT		1 = O S	1 = TOGGLE	1 = PB$_6$ ON	1 = START

|_____ TA _____|

REG	NAME	ALARM	IN	MODE	LOAD	RUN MODE	OUT MODE	PB ON	START
F	CRB	0 = TOD	0	0 φ2	1 = FORCE LOAD	0 = CONT	0 = PULSE	0 = PB$_7$ OFF	0 = STOP
			1	1 = CNT					
			1	0 = TA					
		1 = ALARM	1	1 = CNT-TA	(STROBE)	1 = O S	1 = TOGGLE	1 = PB$_7$ ON	1 = START

|_____ TB _____|

All unused register bits are unaffected by a write and are forced to zero on a read.

6566/6567 (VIC-II) Chip Specifications

The 6566/6567 are multi-purpose color video controller devices for use in both computer video terminals and video game applications. Both devices contain 47 control registers which are accessed via a standard 8-bit microprocessor bus (65XX) and will access up to 16K of memory for display information. The various operating modes and options within each mode are described.

Character Display Mode

In the character display mode, the 6566/6567 fetches CHARACTER POINTERs from the VIDEO MATRIX area of memory and translates the pointers to character dot location addresses in the 2048 byte CHARACTER BASE area of memory. The video matrix is comprised of 1000 consecutive locations in memory which each contain an eight-bit character pointer. The location of the video matrix within memory is defined by VM13-VM10 in register 24($18) which are used as the 4 MSB of the video matrix

address. The lower order 10 bits are provided by an internal counter (VC3-VC0) which steps through the 1000 character locations. Note that the 6566/6567 provides 14 address outputs; therefore, additional system hardware may be required for complete system memory decodes.

CHARACTER POINTER ADDRESS

A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00
VM13	VM12	VM11	VM10	VC9	VC8	VC7	VC6	VC5	VC4	VC3	VC2	VC1	VC0

The eight-bit character pointer permits up to 256 different character definitions to be available simultaneously. Each character is an 8×8 dot matrix stored in the character base as eight consecutive bytes. The location of the character base is defined by CB13-CB11 also in register 24 ($18) which are used for the 3 most significant bits (MSB) of the character base address. The 11 lower order addresses are formed by the 8-bit character pointer from the video matrix (D7-D0) which selects a particular character, and a 3-bit raster counter (RC2-RC0) which selects one of the eight character bytes. The resulting characters are formatted as 25 rows of 40 characters each. In addition to the 8-bit character pointer, a 4-bit COLOR NYBBLE is associated with each video matrix location (the video matrix memory must be 12 bits wide) which defines one of sixteen colors for each character.

CHARACTER DATA ADDRESS

A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00
CB13	CB12	CB11	D7	D6	D5	D4	D3	D2	D1	D0	RC2	RC1	RC0

Standard Character Mode (MCM = BMM = ECM = 0)

In the standard character mode, the 8 sequential bytes from the character base are displayed directly on the 8 lines in each character region. A "0" bit causes the background #0 color (from register 33 ($21)) to be displayed while the color selected by the color nybble (foreground) is displayed for a "1" bit (see Color Code Table).

FUNCTION	CHARACTER BIT	COLOR DISPLAYED
Background	0	Background #0 color (register 33 ($21))
Foreground	1	Color selected by 4-bit color nybble

Therefore, each character has a unique color determined by the 4-bit color nybble (1 of 16) and all characters share the common background color.

Multi-Color Character Mode (MCM = 1, BMM = ECM = 0)

Multi-color mode provides additional color flexibility allowing up to four colors within each character but with reduced resolution. The multi-color mode is selected by setting the MCM bit in register 22 ($16) to "1," which causes the dot data stored in the character base to be interpreted in a different manner. If the MSB of the color nybble is a "0," the character will be displayed as described in standard character mode, allowing the two modes to be inter-mixed (however, only the lower order 8 colors are available) When the MSB of the color nybble is a "1" (if MCM:MSB(CM) = 1) the character bits are interpreted in the multi-color mode:

FUNCTION	CHARACTER BIT PAIR	COLOR DISPLAYED
Background	00	Background #0 Color (register 33 ($21))
Background	01	Background #1 Color (register 34 ($22))
Foreground	10	Background #2 Color (register 35 ($23))
Foreground	11	Color specified by 3 LSB of color nybble

Since two bits are required to specify one dot color, the character is now displayed as a 4 × 8 matrix with each dot twice the horizontal size as in standard mode. Note, however, that each character region can now contain 4 different colors, two as foreground and two as background (see MOB priority).

Extended Color Mode (ECM = 1, BMM = MCM = 0)

The extended color mode allows the selection of individual background colors for each character region with the normal 8 × 8 character resolution. This mode is selected by setting the ECM bit of register 17 ($11) to "1." The character dot data is displayed as in the standard mode (foreground color determined by the color nybble is displayed for a "1" data bit), but the 2 MSB of the character pointer are used to select the background color for each character region as follows:

CHAR. POINTER MS BIT PAIR	BACKGROUND COLOR DISPLAYED FOR 0 BIT
00	Background #0 color (register 33 ($21))
01	Background #1 color (register 34 ($22))
10	Background #2 color (register 35 ($23))
11	Background #3 color (register 36 ($24))

Since the two MSB of the character pointers are used for color information, only 64 different character definitions are available. The 6566/6567 will force CB10 and CB9 to "0" regardless of the original pointer values, so that only the first 64 character definitions will be accessed. With extended color mode each character has one of sixteen individually defined foreground colors and one of the four available background colors.

NOTE: Extended color mode and multi-color mode should not be enabled simultaneously.

Bit Map Mode

In bit map mode, the 6566/6567 fetches data from memory in a different fashion, so that a one-to-one correspondence exists between each displayed dot and a memory bit. The bit map mode provides a screen resolution of 320H × 200V individually controlled display dots. Bit map mode is selected by setting the BMM bit in register 17 ($11) to a "1." The VIDEO MATRIX is still accessed as in character mode, but the video matrix data is no longer interpreted as character pointers, but rather as color data. The VIDEO MATRIX COUNTER is then also used as an address to fetch the dot data for display from the 8000-byte DISPLAY BASE. The display base address is formed as follows:

A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00
CB13	VC9	VC8	VC7	VC6	VC5	VC4	VC3	VC2	VC1	VC0	RC2	RC1	RC0

VCx denotes the video matrix counter outputs, RCx denotes the 3-bit raster line counter and CB13 is from register 24 ($18). The video matrix counter steps through the same 40 locations for eight raster lines, continuing to the next 40 locations every eighth line, while the raster counter increments once for each horizontal video line (raster line). This addressing results in each eight sequential memory locations being formatted as an 8 × 8 dot block on the video display.

Standard Bit Map Mode (BMM = 1, MCM = 0)

When standard bit map mode is in use, the color information is derived only from the data stored in the video matrix (the color nybble is disregarded). The 8 bits are divided into two 4-bit nybbles which allow two colors to be independently selected in each 8 × 8 dot block. When a bit in the display memory is a "0" the color of the output dot is set by the least significant (lower) nybble (LSN) Similarly, a display memory bit of "1" selects the output color determined by the MSN (upper nybble).

BIT	DISPLAY COLOR
0	Lower nybble of video matrix pointer
1	Upper nybble of video matrix pointer

Multi-Color Bit Map Mode (BMM = MCM = 1)

Multi-colored bit map mode is selected by setting the MCM bit in register 22 ($16) to a "1" in conjunction with the BMM bit. Multi-color mode uses the same memory access sequences as standard bit map mode, but interprets the dot data as follows:

BIT PAIR	DISPLAY COLOR
00	Background #0 color (register 33 ($21))
01	Upper nybble of video matrix pointer
10	Lower nybble of video matrix pointer
11	Video matrix color nybble

Note that the color nybble (DB11–DB8) IS used for the multi-color bit map mode, again, as two bits are used to select one dot color, the horizontal dot size is doubled, resulting in a screen resolution of 160H × 200V. Utilizing multi-color bit map mode, three independently selected colors can be displayed in each 8 × 8 block in addition to the background color.

Movable Object Blocks

The movable object block (MOB) is a special type of character which can be displayed at any one position on the screen without the block constraints inherent in character and bit map mode. Up to 8 unique MOBs can be displayed simultaneously, each defined by 63 bytes in memory which are displayed as a 24 × 21 dot array (shown below). A number of special features make MOBs especially suited for video graphics and game applications.

MOB DISPLAY BLOCK

BYTE	BYTE	BYTE
00	01	02
03	04	05
.	.	.
.	.	.
.	.	.
57	58	59
60	61	62

Enable

Each MOB can be selectively enabled for display by setting its corresponding enable bit (MnE) to "1" in register 21 ($15). If the MnE bit is "0," no MOB operations will occur involving the disabled MOB.

Position

Each MOB is positioned via its X and Y position register (see register map) with a resolution of 512 horizontal and 256 vertical positions. The position of a MOB is determined by the upper-left corner of the array. X locations 23 to 347 ($17–$157) and Y locations 50 to 249 ($32–$F9) are visible. Since not all available MOB positions are entirely visible on the screen, MOBs may be moved smoothly on and off the display screen.

Color

Each MOB has a separate 4-bit register to determine the MOB color. The two MOB color modes are:

STANDARD MOB (MnMC = 0)

In the standard mode, a "0" bit of MOB data allows any background data to show through (transparent) and a "1" bit is displayed as the MOB color determined by the corresponding MOB Color register.

MULTI-COLOR MOB (MnMC = 1)

Each MOB can be individually selected as a multi-color MOB via MnMC bits in the MOB Multi-color register 28 ($1C). When the MnMC bit is "1," the corresponding MOB is displayed in the multi-color mode. In the multi-color mode, the MOB data is interpreted in pairs (similar to the other multi-color modes) as follows:

BIT PAIR	COLOR DISPLAYED
00	Transparent
01	MOB Multi-color #0 (register 37 ($25))
10	MOB Color (registers 39–46 ($27–$2E))
11	MOB Multi-color #1 (register 38 ($26))

Since two bits of data are required for each color, the resolution of the MOB is reduced to 12×21, with each horizontal dot expanded to twice standard size so that the overall MOB size does not change. Note that up to 3 colors can be displayed in each MOB (in addition to transparent) but that two of the colors are shared among all the MOBs in the multi-color mode.

Magnification

Each MOB can be selectively expanded ($2 \times$) in both the horizontal and vertical directions. Two registers contain the control bits (MnXE,MnYE) for the magnification control:

REGISTER	FUNCTION
23 ($17)	Horizontal expand MnXE—"1" = expand; "0" = normal
29 ($1D)	Vertical expand MnYE—"1" = expand; "0" = normal

When MOBs are expanded, no increase in resolution is realized. The same 24×21 array (12×21 if multi-colored) is displayed, but the overall MOB dimension is doubled in the desired direction (the smallest MOB dot may be up to 4× standard dot dimension if a MOB is both multi-colored and expanded).

Priority

The priority of each MOB may be individually controlled with respect to the other displayed information from character or bit map modes. The priority of each MOB is set by the corresponding bit (MnDP) of register 27 ($1B) as follows.

REG BIT	PRIORITY TO CHARACTER OR BIT MAP DATA
0	Non-transparent MOB data will be displayed (MOB in front)
1	Non-transparent MOB data wil be displayed only instead of Bkgd #0 or multi-color bit pair 01 (MOB behind)

MOB—DISPLAY DATA PRIORITY

MnDP = 1	MnDP = 0
MOBn	Foreground
Foreground	MOBn
Background	Background

MOB data bits of "0" ("00" in multi-color mode) are transparent, always permitting any other information to be displayed.

The MOBs have a fixed priority with respect to each other, with MOB 0 having the highest priority and MOB 7 the lowest. When MOB data (except transparent data) of two MOBs are coincident, the data from the lower number MOB will be displayed. MOB vs. MOB data is prioritized before priority resolution with character or bit map data.

Collision Detection

Two types of MOB collision (coincidence) are detected, MOB to MOB collision and MOB to display data collision:

1) A collision between two MOBs occurs when non-transparent output data of two MOBs are coincident. Coincidence of MOB transparent areas will not generate a collision. When a collision occurs, the MOB bits (MnM) in the

MOB–MOB COLLISION register 30 ($1E) will be set to "1" for both colliding MOBs. As a collision between two (or more) MOBs occurs, the MOB–MOB collision bit for each collided MOB will be set. The collision bits remain set until a read of the collision register, when all bits are automatically cleared MOBs collisions are detected even if positioned off-screen.

2) The second type of collision is a MOB–DATA collision between a MOB and foreground display data from the character or bit map modes. The MOB–DATA COLLISION register 31 ($1F) has a bit (MnD) for each MOB which is set to "1" when both the MOB and non-background display data are coincident. Again, the coincidence of only transparent data does not generate a collision For special applications, the display data from the 0–1 multicolor bit pair also does not cause a collision. This feature permits their use as background display data without interfering with true MOB collisions. A MOB–DATA collision can occur off-screen in the horizontal direction if actual display data has been scrolled to an off-screen position (see scrolling). The MOB–DATA COLLISION register also automatically clears when read.

The collision interrupt latches are set whenever the first bit of either register is set to "1." Once any collision bit within a register is set high, subsequent collisions will not set the interrupt latch until that collision register has been cleared to all "0s" by a read.

MOB Memory Access

The data for each MOB is stored in 63 consecutive bytes of memory. Each block of MOB data is defined by a MOB pointer, located at the end of the VIDEO MATRIX. Only 1000 bytes of the video matrix are used in the normal display modes, allowing the video matrix locations 1016–1023 (VM base + $3F8 to VM base + $3FF) to be used for MOB pointers 0–7, respectively. The eight-bit MOB pointer from the video matrix together with the six bits from the MOB byte counter (to address 63 bytes) define the entire 14-bit address field:

A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00
MP7	MP6	MP5	MP4	MP3	MP2	MP1	MP0	MC5	MC4	MC3	MC2	MC1	MC0

Where MPx are the MOB pointer bits from the video matrix and MCx are the internally generated MOB counter bits. The MOB pointers are read from the video matrix at the end of every raster line. When the Y position register of a MOB matches the current raster line count, the actual fetches of MOB data begin. Internal counters automatically step through the 63 bytes of MOB data, displaying three bytes on each raster line.

Other Features
Screen Blanking

The display screen may be blanked by setting the DEN bit in register 17 ($11) to a "0." When the screen is blanked, the entire screen will be filled with the exterior color as in register 32 ($20). When blanking is active, only transparent (Phase 1) memory accesses are required, permitting full processor utilization of the system bus. MOB data, however, will be accessed if the MOBs are not also disabled. The DEN bit must be set to "1" for normal video display.

Row/Column Select

The normal display consists of 25 rows of 40 characters (or character regions) per row. For special display purposes, the display window may be reduced to 24 rows and 38 characters. There is no change in the format of the displayed information, except that characters (bits) adjacent to the exterior border area will now be covered by the border. The select bits operate as follows:

RSEL	NUMBER OF ROWS	CSEL	NUMBER OF COLUMNS
0	24 rows	0	38 columns
1	25 rows	1	40 columns

The RSEL bit is in register 17 ($11) and the CSEL bit is in register 22 ($16). For standard display the larger display window is normally used, while the smaller display window is normally used in conjunction with scrolling.

Scrolling

The display data may be scrolled up to one entire character space in both the horizontal and vertical direction. When used in conjunction with the smaller display window (above), scrolling can be used to create a smooth panning motion of display data while updating the system memory only when a new character row (or column) is required. Scrolling is also used to center a fixed display within the display window.

BITS	REGISTER	FUNCTION
X2,X1,X0	22 ($16)	Horizontal Position
Y2,Y1,Y0	17 ($11)	Vertical Position

Light Pen

The light pen input latches the current screen position into a pair of registers (LPX,LPY) on a low-going edge. The X position register 19 ($13) will contain the 8 MSB of the X position at the time of transition. Since the X position is defined by a 512-state counter (9 bits) resolution to 2 horizontal dots is provided. Similarly, the Y position is latched to its register 20 ($14) but here 8 bits provide single raster resolution within the visible display. The light pen latch may be triggered only once per frame, and subsequent triggers within the same frame will have no effect Therefore, you must take several samples before turning the light pen to the screen (3 or more samples, average), depending upon the characteristics of your light pen.

Raster Register

The raster register is a dual-function register. A read of the raster register 18 ($12) returns the lower 8 bits of the current raster position (the MSB– RC8 is located in register 17 ($11)) The raster register can be interrogated to implement display changes outside the visible area to prevent display flicker. The visible display window is from raster 51 through raster 251 ($033–$0FB). A write to the raster bits (including RC8) is latched for use in an internal raster compare. When the current raster matches the written value, the raster interrupt latch is set.

Interrupt Register

The interrupt register shows the status of the four sources of interrupt. An interrupt latch in register 25 ($19) is set to "1" when an interrupt source has generated an interrupt request. The four sources of interrupt are:

LATCH BIT	ENABLE BIT	WHEN SET
IRST	ERST	Set when (raster count) = (stored raster count)
IMDC	EMDC	Set by MOB–DATA collision register (first collision only)
IMMC	EMMC	Set by MOB–MOB collision register (first collision only)
ILP	ELP	Set by negative transition of LP input (once per frame)
IRQ		Set high by latch set and enabled (invert of IRQ/ output)

To enable an interrupt request to set the IRQ/ output to "0," the corresponding interrupt enable bit in register 26 ($1A) must be set to "1." Once an interrupt latch has been set, the latch may be cleared only by writing a "1" to the desired latch in the interrupt register. This feature allows selective handling of video interrupts without software required to "remember" active interrupts

Dynamic Ram Refresh

A dynamic ram refresh controller is built into the 6566/6567 devices. Five 8-bit row addresses are refreshed every raster line. This rate guarantees a maximum delay of 2.02 ms between the refresh of any single row address in a 128 refresh scheme. (The maximum delay is 3.66 ms in a 256 address refresh scheme.) This refresh is totally transparent to the system, since the refresh occurs during Phase 1 of the system clock. The 6567 generates both RAS/ and CAS/ which are normally connected directly to the dynamic rams. RAS/ and CAS/ are generated for every Phase 2 and every video data access (including refresh) so that external clock generation is not required.

Reset

The reset bit (RES) in register 22 ($16) is not used for normal operation. Therefore it should be set to "0" when initializing the video chip. When set to a "1," the entire operation of the video chip is suspended, including video outputs and sync, memory refresh, and system bus access

Theory of Operation

System Interface

The 6566/6567 video controller devices interact with the system data bus in a special way. A 65XX system requires the system buses only during the Phase 2 (clock high) portion of the cycle. The 6566/6567 devices take advantage of this feature by normally accessing system memory during the Phase 1 (clock low) portion of the clock cycle Therefore, operations such as character data fetches and memory refresh are totally transparent to the processor and do not reduce the processor throughput. The video chips provide the interface control signals required to maintain this bus sharing.

The video devices provide the signal AEC (address enable control) which is used to disable the processor address bus drivers allowing the video device to access the address bus. AEC is active low which permits direct

connection to the AEC input of the 65XX family. The AEC signal is normally activated during Phase 1 so that processor operation is not affected. Because of this bus "sharing," all memory accesses must be completed in ½ cycle. Since the video chips provide a 1-MHz clock (which must be used as system Phase 2), a memory cycle is 500 ns including address setup, data access and, data setup to the reading device.

Certain operations of the 6566/6567 require data at a faster rate than available by reading only during the Phase 1 time; specifically, the access of character pointers from the video matrix and the fetch of MOB data. Therefore, the processor must be disabled and the data accessed during the Phase 2 clock. This is accomplished via the BA (bus available) signal. The BA line is normally high but is brought low during Phase 1 to indicate that the video chip will require a Phase 2 data access. Three Phase-2 times are allowed after BA low for the processor to complete any current memory accesses. On the fourth Phase 2 after BA low, the AEC signal will remain low during Phase 2 as the video chip fetches data. The BA line is normally connected to the RDY input of a 65XX processor. The character pointer fetches occur every eighth raster line during the display window and require 40 consecutive Phase 2 accesses to fetch the video matrix pointers. The MOB data fetches require 4 memory accesses as follows:

PHASE	DATA	CONDITION
1	MOB Pointer	Every raster
2	MOB Byte 1	Each raster while MOB is displayed
1	MOB Byte 2	Each raster while MOB is displayed
2	MOB Byte 3	Each raster while MOB is displayed

The MOB pointers are fetched every other Phase 1 at the end of each raster line. As required, the additional cycles are used for MOB data fetches. Again, all necessary bus control is provided by the 6566/6567 devices.

Memory Interface

The two versions of the video interface chip, 6566 and 6567, differ in address output configurations. The 6566 has thirteen fully decoded addresses for direct connection to the system address bus. The 6567 has multiplexed addresses for direct connection to 64K dynamic RAMs. The least significant address bits, A06-A00, are present on A06-A00 while RAS/ is brought low, while the most significant bits, A13-A08, are present on A05-A00 while CAS/ is brought low. The pins A11-A07 on the 6567 are

REGISTER MAP

ADDRESS	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	DESCRIPTION
00 ($00)	M0X7	M0X6	M0X5	M0X4	M0X3	M0X2	M0X1	M0X0	MOB 0 X-position
01 ($01)	M0Y7	M0Y6	M0Y5	M0Y4	M0Y3	M0Y2	M0Y1	M0Y0	MOB 0 Y-position
02 ($02)	M1X7	M1X6	M1X5	M1X4	M1X3	M1X2	M1X1	M1X0	MOB 1 X-position
03 ($03)	M1Y7	M1Y6	M1Y5	M1Y4	M1Y3	M1Y2	M1Y1	M1Y0	MOB 1 Y-position
04 ($04)	M2X7	M2X6	M2X5	M2X4	M2X3	M2X2	M2X1	M2X0	MOB 2 X-position
05 ($05)	M2Y7	M2Y6	M2Y5	M2Y4	M2Y3	M2Y2	M2Y1	M2Y0	MOB 2 Y-position
06 ($06)	M3X7	M3X6	M3X5	M3X4	M3X3	M3X2	M3X1	M3X0	MOB 3 X-position
07 ($07)	M3Y7	M3Y6	M3Y5	M3Y4	M3Y3	M3Y2	M3Y1	M3Y0	MOB 3 Y-position
08 ($08)	M4X7	M4X6	M4X5	M4X4	M4X3	M4X2	M4X1	M4X0	MOB 4 X-position
09 ($09)	M4Y7	M4Y6	M4Y5	M4Y4	M4Y3	M4Y2	M4Y1	M4Y0	MOB 4 Y-position
10 ($0A)	M5X7	M5X6	M5X5	M5X4	M5X3	M5X2	M5X1	M5X0	MOB 5 X-position
11 ($0B)	M5Y7	M5Y6	M5Y5	M5Y4	M5Y3	M5Y2	M5Y1	M5Y0	MOB 5 Y-position
12 ($0C)	M6X7	M6X6	M6X5	M6X4	M6X3	M6X2	M6X1	M6X0	MOB 6 X-position
13 ($0D)	M6Y7	M6Y6	M6Y5	M6Y4	M6Y3	M6Y2	M6Y1	M6Y0	MOB 6 Y-position
14 ($0E)	M7X7	M7X6	M7X5	M7X4	M7X3	M7X2	M7X1	M7X0	MOB 7 X-position
15 ($0F)	M7Y7	M7Y6	M7Y5	M7Y4	M7Y3	M7Y2	M7Y1	M7Y0	MOB 7 Y-position
16 ($10)	M7X8	M6X8	M5X8	M4X8	M3X8	M2X8	M1X8	M0X8	MSB of X-position
17 ($11)	RC8	ECM	BMM	DEN	RSEL	Y2	Y1	Y0	See text
18 ($12)	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0	Raster register
19 ($13)	LPX8	LPX7	LPX6	LPX5	LPX4	LPX3	LPX2	LPX1	Light Pen X
20 ($14)	LPY7	LPY6	LPY5	LPY4	LPY3	LPY2	LPY1	LPY0	Light Pen Y
21 ($15)	M7E	M6E	M5E	M4E	M3E	M2E	M1E	M0E	MOB Enable
22 ($16)	—	—	RES	MCM	CSEL	X2	X1	X0	See text
23 ($17)	M7YE	M6YE	M5YE	M4YE	M3YE	M2YE	M1YE	M0YE	MOB Y-expand
24 ($18)	VM13	VM12	VM11	VM10	CB13	CB12	CB11	—	Memory Pointers
25 ($19)	IRQ	—	—	—	ILP	IMMC	IMBC	IRST	Interrupt Register
26 ($1A)	—	—	—	—	ELP	EMMC	EMBC	ERST	Enable Interrupt

			Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
27	($1B)	M7DP	M6DP	M5DP	M4DP	M3DP	M2DP	M1DP	M0DP	MOB-DATA Priority
28	($1C)	M7MC	M6MC	M5MC	M4MC	M3MC	M2MC	M1MC	M0MC	MOB Multicolor Sel
29	($1D)	M7XE	M6XE	M5XE	M4XE	M3XE	M2XE	M1XE	M0XE	MOB X-expand
30	($1E)	M7M	M6M	M5M	M4M	M3M	M2M	M1M	M0M	MOB-MOB Collision
31	($1F)	M7D	M6D	M5D	M4D	M3D	M2D	M1D	M0D	MOB-DATA Collision
32	($20)	—	—	—	—	EC3	EC2	EC1	EC0	Exterior Color
33	($21)	—	—	—	—	BOC3	BOC2	BOC1	BOC0	Bkgd #0 Color
34	($22)	—	—	—	—	B1C3	B1C2	B1C1	B1C0	Bkgd #1 Color
35	($23)	—	—	—	—	B2C3	B2C2	B2C1	B2C0	Bkgd #2 Color
36	($24)	—	—	—	—	B3C3	B3C2	B3C1	B3C0	Bkgd #3 Color
37	($25)	—	—	—	—	MM03	MM02	MM01	MM00	MOB Multicolor #0
38	($26)	—	—	—	—	MM13	MM12	MM11	MM10	MOB Multicolor #1
39	($27)	—	—	—	—	M0C3	M0C2	M0C1	M0C0	MOB 0 Color
40	($28)	—	—	—	—	M1C3	M1C2	M1C1	M1C0	MOB 1 Color
41	($29)	—	—	—	—	M2C3	M2C2	M2C1	M2C0	MOB 2 Color
42	($2A)	—	—	—	—	M3C3	M3C2	M3C1	M3C0	MOB 3 Color
43	($2B)	—	—	—	—	M4C3	M4C2	M4C1	M4C0	MOB 4 Color
44	($2C)	—	—	—	—	M5C3	M5C2	M5C1	M5C0	MOB 5 Color
45	($2D)	—	—	—	—	M6C3	M6C2	M6C1	M6C0	MOB 6 Color
46	($2E)	—	—	—	—	M7C3	M7C2	M7C1	M7C0	MOB 7 Color

NOTE: A dash indicates a no connect. All no connects are read as a "1".

COLOR CODES

D4	D3	D1	D0	HEX	DEC	COLOR
0	0	0	0	0	0	BLACK
0	0	0	1	1	1	WHITE
0	0	1	0	2	2	RED
0	0	1	1	3	3	CYAN
0	1	0	0	4	4	PURPLE
0	1	0	1	5	5	GREEN
0	1	1	0	6	6	BLUE
0	1	1	1	7	7	YELLOW
1	0	0	0	8	8	ORANGE
1	0	0	1	9	9	BROWN
1	0	1	0	A	10	LT RED
1	0	1	1	B	11	DARK GRAY
1	1	0	0	C	12	MED GRAY
1	1	0	1	D	13	LT GREEN
1	1	1	0	E	14	LT BLUE
1	1	1	1	F	15	LT GRAY

static address outputs to allow direct connection of these bits to a conventional 16K (2K × 8) ROM. (The lower order addresses require external latching.)

6581 Sound Interface Device (SID) Chip Specifications

Concept

The 6581 Sound Interface Device (SID) is a single-chip, 3-voice electronic music synthesizer/sound effects generator compatible with the 65XX and similar microprocessor families. SID provides wide-range, high-resolution control of pitch (frequency), tone color (harmonic content), and dynamics (volume). Specialized control circuitry minimizes software overhead, facilitating use in arcade/home video games and low-cost musical instruments.

Features

- 3 TONE OSCILLATORS
  Range: 0–4 kHz

- 4 WAVEFORMS PER OSCILLATOR
  Triangle, Sawtooth,
  Variable Pulse, Noise
- 3 AMPLITUDE MODULATORS
  Range: 48 dB
- 3 ENVELOPE GENERATORS
  Exponential response
  Attack Rate 2 ms—8 s
  Decay Rate· 6 ms—24 s
  Sustain Level. 0—peak volume
  Release Rate 6 ms—24 s
- OSCILLATOR SYNCHRONIZATION
- RING MODULATION

Description

The 6581 consists of three synthesizer "voices" which can be used in-dependently or in conjunction with each other (or external audio sources) to create complex sounds. Each voice consists of a Tone Oscillator/Wave-form Generator, an Envelope Generator and an Amplitude Modulator. The Tone Oscillator controls the pitch of the voice over a wide range. The Oscillator produces four waveforms at the selected frequency, with the unique harmonic content of each waveform providing simple control of tone color. The volume dynamics of the oscillator are controlled by the Amplitude Modulator under the direction of the Envelope Generator. When triggered, the Envelope Generator creates an amplitude envelope with programmable rates of increasing and decreasing volume. In addition to the three voices, a programmable Filter is provided for generating complex, dynamic tone colors via subtractive synthesis.

SIS allows the microprocessor to read the changing output of the third Oscillator and third Envelope Generator. These outputs can be used as a source of modulation information for creating vibrator, frequency/filter sweeps and similar effects. The third oscillator can also act as a random number generator for games. Two A/D converters are provided for inter-facing SID with potentiometers. These can be used for "paddles" in a game environment or as front panel controls in a music synthesizer. SID can process external audio signals, allowing multiple SID chips to be daisy-chained or mixed in complex polyphonic systems.

SID Control Registers

There are 29 eight-bit registers in SID which control the generation of sound. These registers are either WRITE-only or READ-only and are listed below in Table 1.

SID Register Description

Voice 1

FREQ LO/FREQ HI (Registers 00,01)

Together these registers form a 16-bit number which linearly controls the frequency of Oscillator 1. The frequency is determined by the following equation:

$$F_{out} = (F_n \times F_{clk}/16777216) \text{ Hz}$$

Where F_n is the 16-bit number in the Frequency registers and F_{clk} is the system clock applied to the $\phi 2$ input (pin 6). For a standard 1.0-MHz clock, the frequency is given by:

$$F_{out} = (F_n \times 0.059604645) \text{ Hz}$$

A complete table of values for generating 8 octaves of the equally tempered musical scale with concert A (440 Hz) tuning is provided in Appendix E. It should be noted that the frequency resolution of SID is sufficient for any tuning scale and allows sweeping from note to note (portamento) with no discernable frequency steps.
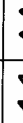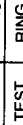
PW LO/PW HI (Registers 02,03)
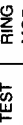
Together these registers form a 12-bit number (bits 4–7 of PW HI are not used) which linearly controls the Pulse Width (duty cycle) of the Pulse waveform on Oscillator 1. The pulse width is determined by the following equation:

$$PW_{out} = (PW_n/40.95)\%$$

Where PW_n is the 12-bit number in the Pulse Width registers.

The pulse width resolution allows the width to be smoothly swept with no discernable stepping. Note that the Pulse waveform on Oscillator 1 must be selected in order for the Pulse Width registers to have any audible

A4	A3	A2	A1	A0	REG # (HEX)	D7	D6	D5	D4	D3	D2	D1	D0	REG NAME	REG TYPE
														Voice 1	
0	0	0	0	0	00	F_7	F_6	F_5	F_4	F_3	F_2	F_1	F_0	FREQ LO	WRITE-ONLY
0	0	0	0	1	01	F_{15}	F_{14}	F_{13}	F_{12}	F_{11}	F_{10}	F_9	F_8	FREQ HI	WRITE-ONLY
0	0	0	1	0	02	PW_7	PW_6	PW_5	PW_4	PW_3	PW_2	PW_1	PW_0	PW-LO	WRITE-ONLY
0	0	0	1	1	03	—	—	—	—	PW_{11}	PW_{10}	PW_9	PW_8	PW HI	WRITE-ONLY
0	0	1	0	0	04	NOISE	⊓⊔	∿	∧	TEST	RING MOD	SYNC	GATE	CONTROL REG	WRITE-ONLY
0	0	1	0	1	05	ATK_3	ATK_2	ATK_1	ATK_0	DCY_3	DCY_2	DCY_1	DCY_0	ATTACK/DECAY	WRITE-ONLY
0	0	1	1	0	06	STN_3	STN_2	STN_1	STN_0	RLS_3	RLS_2	RLS_1	RLS_0	SUSTAIN/RELEASE	WRITE-ONLY
														Voice 2	
0	0	1	1	1	07	F_7	F_6	F_5	F_4	F_3	F_2	F_1	F_0	FREQ LO	WRITE-ONLY
0	1	0	0	0	08	F_{15}	F_{14}	F_{13}	F_{12}	F_{11}	F_{10}	F_9	F_8	FREQ HI	WRITE-ONLY
0	1	0	0	1	09	PW_7	PW_6	PW_5	PW_4	PW_3	PW_2	PW_1	PW_0	PW LO	WRITE-ONLY
0	1	0	1	0	0A	—	—	—	—	PW_{11}	PW_{10}	PW_9	PW_8	PW HI	WRITE-ONLY
0	1	0	1	1	0B	NOISE	⊓⊔	∿	∧	TEST	RING MOD	SYNC	GATE	CONTROL REG	WRITE-ONLY
0	1	1	0	0	0C	ATK_3	ATK_2	ATK_1	ATK_0	DCY_3	DCY_2	DCY_1	DCY_0	ATTACK/DECAY	WRITE-ONLY
0	1	1	0	1	0D	STN_3	STN_2	STN_1	STN_0	RLS_3	RLS_2	RLS_1	RLS_0	SUSTAIN/RELEASE	WRITE-ONLY
														Voice 3	
0	1	1	1	0	0E	F_7	F_6	F_5	F_4	F_3	F_2	F_1	F_0	FREQ LO	WRITE-ONLY
0	1	1	1	1	0F	F_{15}	F_{14}	F_{13}	F_{12}	F_{11}	F_{10}	F_9	F_8	FREQ HI	WRITE-ONLY
1	0	0	0	0	10	PW_7	PW_6	PW_5	PW_4	PW_3	PW_2	PW_1	PW_0	PW LO	WRITE-ONLY
1	0	0	0	1	11	—	—	—	—	PW_{11}	PW_{10}	PW_9	PW_8	PW HI	WRITE-ONLY
1	0	0	1	0	12	NOISE	⊓⊔	∿	∧	TEST	RING MOD	SYNC	GATE	CONTROL REG	WRITE-ONLY
1	0	0	1	1	13	ATK_3	ATK_2	ATK_1	ATK_0	DCY_3	DCY_2	DCY_1	DCY_0	ATTACK/DECAY	WRITE-ONLY
1	0	1	0	0	14	STN_3	STN_2	STN_1	STN_0	RLS_3	RLS_2	RLS_1	RLS_0	SUSTAIN/RELEASE	WRITE-ONLY

Filter

Dec	Bin	Hex	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Register	Access
21	10101	15	—	—	—	—	—	FC$_2$	FC$_1$	FC$_0$	FC LO	WRITE-ONLY
22	10110	16	FC$_{10}$	FC$_9$	FC$_8$	FC$_7$	FC$_6$	FC$_5$	FC$_4$	FC$_3$	FC HI	WRITE-ONLY
23	10111	17	RES$_3$	RES$_2$	RES$_1$	RES$_0$	FILTEX	FILT 3	FILT 2	FILT 1	RES/FILT	WRITE-ONLY
24	11000	18	3 OFF	HP	BP	LP	VOL$_3$	VOL$_2$	VOL$_1$	VOL$_0$	MODE/VOL	WRITE-ONLY

Misc

Dec	Bin	Hex	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Register	Access
25	11001	19	PX$_7$	PX$_6$	PX$_5$	PX$_4$	PX$_3$	PX$_2$	PX$_1$	PX$_0$	POT X	READ-ONLY
26	11010	1A	PY$_7$	PY$_6$	PY$_5$	PY$_4$	PY$_3$	PY$_2$	PY$_1$	PY$_0$	POT Y	READ-ONLY
27	11011	1B	O$_7$	O$_6$	O$_5$	O$_4$	O$_3$	O$_2$	O$_1$	O$_0$	aOSC$_3$/RANDOM	READ-ONLY
28	11100	1C	E$_7$	E$_6$	E$_5$	E$_4$	E$_3$	E$_2$	E$_1$	E$_0$	ENV$_3$	READ-ONLY

Figure I.4

effect. A value of 0 or 4095 ($FFF) in the Pulse Width registers will produce a constant DC output, while a value of 2048 ($800) will produce a square wave.

CONTROL REGISTER (Register 04)

This register contains eight control bits which select various options on Oscillator 1.

Gate (Bit 0): The GATE bit controls the Envelope Generator for Voice 1. When this bit is set to a one, the Envelope Generator is Gated (triggered) and the ATTACK/DECAY/SUSTAIN cycle is initiated. When the bit is reset to a zero, the RELEASE cycle begins. The Envelope Generator controls the amplitude of Oscillator 1 appearing at the audio output, therefore, the GATE bit must be set (along with suitable envelope parameters) for the selected output of Oscillator 1 to be audible. A detailed discussion of the Envelope Generator can be found at the end of this Appendix.

SYNC (Bit 1): The SYNC bit, when set to a one, synchronizes the fundamental frequency of Oscillator 1 with the fundamental frequency of Oscillator 3, producing "Hard Sync" effects.

Varying the frequency of Oscillator 1 with respect to Oscillator 3 produces a wide range of complex harmonic structures from Voice 1 at the frequency of Oscillator 3. In order for sync to occur, Oscillator 3 must be set to some frequency other than zero but preferably lower than the frequency of Oscillator 1. No other parameters of Voice 3 have any effect on sync.

RING MOD (Bit 2): The RING MOD bit, when set to a one, replaces the Triangle waveform output of Oscillator 1 with a "Ring Modulated" combination of Oscillators 1 and 3. Varying the frequency of Oscillator 1 with respect to Oscillator 3 produces a wide range of non-harmonic overtone structures for creating bell or gong sounds and for special effects. In order for ring modulation to be audible, the Triangle waveform of Oscillator 1 must be selected and Oscillator 3 must be set to some frequency other than zero. No other parameters of Voice 3 have any effect on ring modulation.

TEST (Bit 3): The TEST bit, when set to a one, resets and locks Oscillator 1 at zero until the TEST bit is cleared. The Noise waveform output of Oscillator 1 is also reset and the Pulse waveform output is held at a DC level. Normally this bit is used for testing purposes, however, it can be used to synchronize Oscillator 1 to external events, allowing the generation of highly complex waveforms under real-time software control.

(Bit 4): When set to a one, the Triangle waveform output of Oscillator 1 is selected. The Triangle waveform is low in harmonics and has a mellow, flute-like quality.

(Bit 5): When set to a one, the Pulse waveform output of Oscillator 1 is selected. The Sawtooth waveform is rich in even and odd harmonics and has a bright, brassy quality.

(Bit 6): When set to a one, the Pulse waveform of Oscillator 1 is selected. The harmonic content of this waveform can be adjusted by the Pulse Width registers, producing tone qualities ranging from a bright, hollow square wave to a nasal, reedy pulse. Sweeping the pulse width in real-time produces a dynamic "phasing" effect which adds a sense of motion to the sound. Rapidly jumping between different pulse widths can produce interesting harmonic sequences

NOISE (Bit 7): When set to a one, the Noise output waveform of Oscillator 1 is selected. This output is a random signal which changes at the frequency of Oscillator 1. The sound quality can be varied from a low rumbling to hissing white noise via the Oscillator 1 Frequency registers. Noise is useful in creating explosions, gunshots, jet engines, wind, surf and other un-pitched sounds, as well as snare drums and cymbals. Sweeping the oscillator frequency with Noise selected produces a dramatic rushing effect.

One of the output waveforms must be selected for Oscillator 1 to be audible, however, it is NOT necessary to de-select waveforms to silence the output of Voice 1. The amplitude of Voice 1 at the final output is a function of the Envelope Generator only.

> **NOTE:** The oscillator output waveforms are NOT additive. If more than one output waveform is selected simultaneously, the result will be a logical ANDing of the waveforms. Although this technique can be used to generate additional waveforms beyond the four listed above, it must be used with care. If any other waveform is selected while Noise is on, the Noise output can "lock up." If this occurs, the Noise output will remain silent until reset by the TEST bit or by bringing RES (pin 5) low.

ATTACK/DECAY (Register 05)

Bits 4–7 of this register (ATK0–ATK3) select 1 of 16 ATTACK rates for the Voice 1 Envelope Generator. The ATTACK rate determines how rapidly the output of Voice 1 rises from zero to peak amplitude when the Envelope Generator is Gated. The 16 ATTACK rates are listed in Table 2.

Bits 0–3 (DCY0—DCY3) select 1 of 16 DECAY rates for the Envelope Generator. The DECAY cycle follows the ATTACK cycle and the DECAY rate determines how rapidly the output falls from the peak amplitude to the selected SUSTAIN level The 16 DECAY rates are listed in Table 2.

SUSTAIN/RELEASE (Register 06)

Bits 4–7 of this register (STN0—STN3) select 1 of 16 SUSTAIN levels for the Envelope Generator. The SUSTAIN cycle follows the DECAY cycle and the output of Voice 1 will remain at the selected SUSTAIN amplitude as long as the Gate bit remains set. The SUSTAIN levels range from zero

Table 2. Envelope Rates

VALUE	ATTACK RATE	DECAY/RELEASE RATE
DEC (HEX)	(Time/Cycle)	(Time/Cycle)
0 (0)	2 ms	6 ms
1 (1)	8 ms	24 ms
2 (2)	16 ms	48 ms
3 (3)	24 ms	72 ms
4 (4)	38 ms	114 ms
5 (5)	56 ms	168 ms
6 (6)	68 ms	204 ms
7 (7)	80 ms	240 ms
8 (8)	100 ms	300 ms
9 (9)	250 ms	750 ms
10 (A)	500 ms	1.5 s
11 (B)	800 ms	2.4 s
12 (C)	1 s	3 s
13 (D)	3 s	9 s
14 (E)	5 s	15 s
15 (F)	8 s	24 s

NOTE: Envelope rates are based on a 1.0-MHz ϕ2 clock. For other ϕ2 frequencies, multiply the given rate by 1 MHz/ϕ2. The rates refer to the amount of time per cycle. For example, given an ATTACK value of 2, the ATTACK cycle would take 16 ms to rise from zero to peak amplitude. The DECAY/RELEASE rates refer to the amount of time these cycles would take to fall from peak amplitude to zero.

to peak amplitude in 16 linear steps, with a SUSTAIN value of 0 selecting zero amplitude and a SUSTAIN value of 15 ($F) selecting the peak amplitude. A SUSTAIN value of 8 would cause Voice 1 to SUSTAIN at an amplitude one-half the peak amplitude reached by the ATTACK cycle.

Bits 0–3 (RLS0–RLS3) select 1 of 16 RELEASE rates for the Envelope Generator. The RELEASE cycle follows the SUSTAIN cycle when the Gate bit is reset to zero. At this time, the output of Voice 1 will fall from the SUSTAIN amplitude to zero amplitude at the selected RELEASE rate. The 16 RELEASE rates are identical to the DECAY rates.

NOTE: The cycling of the Envelope Generator can be altered at any point via the Gate bit. The Envelope Generator can be Gated and Released without restriction. For example, if the Gate bit is reset before the envelope has finished the ATTACK cycle, the RELEASE cycle will immediately begin, starting from whatever amplitude had been reached. If the envelope is then gated again (before the RELEASE cycle has reached zero amplitude), another ATTACK cycle will begin, starting from whatever amplitude had been reached. This technique can be used to generate complex amplitude envelopes via real-time software control

---

Voice 2

Registers 07–$0D control Voice 2 and are functionally identical to registers 00–06 with these exceptions:

1) When selected, SYNC synchronizes Oscillator 2 with Oscillator 1.
2) When selected, RING MOD replaces the Triangle output of Oscillator 2 with the ring modulated combination of Oscillators 2 and 1

Voice 3

Registers $0E–$14 control Voice 3 and are functionally identical to registers 00–06 with these exceptions:

1) When selected, SYNC synchronizes Oscillator 3 with Oscillator 2.
2) When selected, RING MOD replaces the Triangle output of Oscillator 3 with the ring modulated combination of Oscillators 3 and 2

Typical operation of a voice consists of selecting the desired parameters: frequency, waveform, effects (SYNC, RING MOD) and envelope rates, then gating the voice whenever the sound is desired. The sound can be sustained for any length of time and terminated by clearing the Gate bit.

Each voice can be used separately, with independent parameters and gating, or in unison to create a single, powerful voice. When used in unison, a slight detuning of each oscillator or tuning to musical intervals creates a rich, animated sound.

Filter

FC LO/FC HI (Registers $15,$16)

Together these registers form an 11-bit number (bits 3–7 of FC LO are not used) which linearly controls the Cutoff (or Center) Frequency of the programmable Filter. The approximate Cutoff Frequency ranges from 30 Hz to 12 KHz.

RES/FILT (Register $17)

Bits 4–7 of this register (RES0–RES3) control the resonance of the filter. Resonance is a peaking effect which emphasizes frequency components at the Cutoff Frequency of the Filter, causing a sharper sound. There are 16 resonance settings ranging linearly from no resonance (0) to maximum resonance (15 or $F). Bits 0–3 determine which signals will be routed through the Filter:

FILT 1 (Bit 0): When set to a zero, Voice 1 appears directly at the audio output and the Filter has no effect on it. When set to a one, Voice 1 will be processed through the Filter and the harmonic content of Voice 1 will be altered according to the selected Filter parameters.

FILT 2 (Bit 1): Same as bit 0 for Voice 2.

FILT 3 (Bit 2): Same as bit 0 for Voice 3.

FILTEX (Bit 3): Same as bit 0 for External audio input (pin 26).

MODE VOL (Register $18)

Bits 4–7 of this register select various Filter mode and output options:

LP (Bit 4): When set to a one, the Low-Pass output of the Filter is selected and sent to the audio output. For a given Filter input signal, all frequency components below the Filter Cutoff Frequency are passed unaltered, while all frequency components above the Cutoff are attenuated at a rate of 12 dB/Octave. The Low-Pass mode produces full-bodied sounds.

BP (Bit 5): Same as bit 4 for the Bandpass output. All frequency components above and below the Cutoff are attenuated at a rate of 6 dB/Octave The Bandpass mode produces thin, open sounds.

HP (Bit 6): Same as bit 4 for the High-Pass output. All frequency components above the Cutoff are passed unaltered, while all frequency components below the Cutoff are attenuated at a rate of 12 dB/Octave. The High-Pass mode produces tinny, buzzy sounds.

3 OFF (Bit 7): When set to a one, the output of voice 3 is disconnected from the direct audio path. Setting Voice 3 to bypass the Filter (FILT 3 = 0) and setting 3 OFF to a one prevents Voice 3 from reaching the audio output. This allows Voice 3 to be used for modulation purposes without any undesirable output.

> **NOTE:** The Filter output modes ARE additive and multiple Filter modes may be selected simultaneously. For example, both LP and HP modes can be selected to produce a Notch (or Band Reject) Filter response. In order for the Filter to have any audible effect, at least one Filter output must be selected and at least one Voice must be routed through the Filter. The Filter is, perhaps, the most important element in SID as it allows the generation of complex tone colors via subtractive synthesis (the Filter is used to eliminate specific frequency components from a harmonically rich input signal). The best results are achieved by varying the Cutoff Frequency in real-time.

Bits 0–3 (VOL0–VOL3) select 1 of 16 overall Volume levels for the final composite audio output The output volume levels range from no output (0) to maximum volume (15 or $F) in 16 linear steps. This control can be used as a static volume control for balancing levels in multi-chip systems or for creating dynamic volume effects, such as Tremolo. Some Volume level other than zero must be selected in order for SID to produce any sound.

Miscellaneous

POTX (Register $19)

This register allows the microprocessor to read the position of the potentiometer tied to POTX (pin 24), with values ranging from 0 at minimum resistance, to 255 ($FF) at maximum resistance. The value is always valid and is updated every 512 φ2 clock cycles. See the Pin Description section for information on pot and capacitor values.

POTY (Register $1A)

Same as POTX for the pot tied to POTY (pin 23).

OSC 3/RANDOM (Register $1B)

This register allows the microprocessor to read the upper 8 output bits of Oscillator 3. The character of the numbers generated is directly related to the waveform selected. If the Sawtooth waveform of Oscillator 3 is selected, this register will present a series of numbers incrementing from 0 to 255 ($FF) at a rate determined by the frequency of Oscillator 3. If the Triangle waveform is selected, the output will increment from 0 up to 255, then decrement down to 0. If the Pulse waveform is selected, the output will jump between 0 and 255. Selecting the Noise waveform will produce a series of random numbers, therefore, this register can be used as a random number generator for games. There are numerous timing and sequencing applications for the OSC 3 register, however, the chief function is probably that of a modulation generator. The numbers generated by this register can be added, via software, to the Oscillator or Filter Frequency registers or the Pulse Width registers in real-time. Many dynamic effects can be generated in this manner. Siren-like sounds can be created by adding the OSC 3 Sawtooth output to the frequency control of another oscillator. Synthesizer "Sample and Hold" effects can be produced by adding the OSC 3 Noise output to the Filter Frequency control registers. Vibrato can be produced by setting Oscillator 3 to a frequency around 7 Hz and adding the OSC 3 Triangle output (with proper scaling) to the Frequency control of another oscillator. An unlimited range of effects are available by altering the frequency of Oscillator 3 and scaling the OSC 3 output. Normally, when Oscillator 3 is used for modulation, the audio output of Voice 3 should be eliminated (3 OFF = 1).

ENV 3 (Register $1C)

Same as OSC 3, but this register allows the microprocessor to read the output of the Voice 3 Envelope Generator. This output can be added to the Filter Frequency to produce harmonic envelopes, WAH-WAH, and similar effects. "Phaser" sounds can be created by adding this output to the frequency control registers of an oscillator. The Voice 3 Envelope Generator must be Gated in order to produce any output from this register. The OSC 3 register, however, always reflects the changing output of the oscillator and is not affected in any way by the Envelope Generator.

6525 Tri-Port Interface

Concept

The 6525 TRI-PORT Interface (TPI) is designed to simplify the implementation of complex I/O operations in microcomputer systems. It combines two dedicated 8-bit I/O ports with a third 8-bit port programmable for either normal I/O operations or priority interrupt/handshaking control. Depending on the mode selected, the 6525 can provide 24 individually programmable I/O lines or 16 I/O lines, 2 handshake lines and 5 priority interrupt inputs.

6525 Addressing

6525 REGISTERS/(Direct Addressing)

*000	R0	PRA—Port Register A
001	R1	PRB—Port Register B
010	R2	PRC—Port Register C
011	R3	DDRA—Data Direction Register A
100	R4	DDRB—Data Direction Register B
101	R5	DDRC—Data Direction Register C/Interrupt Mask Register
110	R6	CR—Control Register
111	R7	AIR—Active Interrupt Register

*NOTE. RS2, RS1, RS0 respectively

6525 Control Registers

CR	CB_1	CB_0	CA_1	CA_0	IE_4	IE_3	IP	MC

AIR			A_4	A_3	A_2	A_1	A_0

DDRC When MC = 1			M_4	M_3	M_2	M_1	M_0

PRC When MC = 1	CB	CA	$\overline{IRQ}$	I_4	I_3	I_2	I_1	I_0

CA, CB Functional Description

The CA, CB lines are outputs used in the same fashion as the CA_2 and CB_2 output of the 6520.

CA Output Modes

CA_1	CA_0	MODE	DESCRIPTION
0	0	"Handshake" on Read	CA is set high on an active transition of the I_3 interrupt input signal and set low by a microprocessor "Read A Data" operation This allows positive control of data transfers from the peripheral device to the microprocessor.
0	1	Pulse Output	CA goes low for IMS after a "Read A Data" operation This pulse can be used to signal the peripheral device that data was taken.
1	0	Manual Output	CA set low
1	1	Manual Output	CA set high.

CB Output Modes

CB_1	CB_0	MODE	DESCRIPTION
0	0	"Handshake" on Write	CB is set low on microprocessor "Write B Data" operation and is set high by an active transition of the I_4 interrupt input signal. This allows positive control of data transfers from the microprocessor to the peripheral device.
0	1	Pulse Output	CB goes low for IMS after a microprocessor "Write B Data" operation. This can be used to signal the peripheral device that data is available.
1	0	Manual Output	CB set low.
1	1	Manual Output	CB set high.

INTERRUPT MASK REGISTER DESCRIPTION

When the Interrupt Mode is selected (MC = 1), the Data Direction Register for Port C (DDRC) is used to enable or disable a corresponding interrupt input. For example: If M_0 = 0 then I_0 is disabled and any I_0 interrupt latched in the interrupt latch register will not be transferred to the AIR and will not cause IRQ to go low. The interrupt latch can be cleared by writing a zero to the appropriate I bit in PRC.

PORT REGISTER C DESCRIPTION

Port Register C (PRC) can operate in two modes. The mode is controlled by bit MC in register CR. When MC = 0, PRC is a standard I/O port, operating identically to PRA & PRB. If MC = 1, then port register C is used for handshaking and priority interrupt input and output.

PRC When MC = 0:

PC_7	PC_6	PC_5	PC_4	PC_3	PC_2	PC_1	PC_0

PRC When MC = 1:

CB	CA	$\overline{IRQ}$	I_4	I_3	I_2	I_1	I_0

INTERRUPT EDGE CONTROL

Bits IE_4 and IE_3 in the control register (CR) are used to determine the active edge which will be recognized by the interrupt latch.

If IE_4 (IE_3) = 0 then I_4 (I_3) latch will be set on a negative transition of I_4 (I_3) input.

If IE_4 (IE_3) = 1 then I_4 (I_3) latch will be set on a positive transition of the I_4 (I_3) input.

All other interrupt latches (I_2, I_1, I_0) are set on a negative transition of the corresponding interrupt input.

I_4	I_3	I_2	I_1	I_0

Interrupt Latch Register

Clears on Read of AIR Using Following Equation

$ILR \leftarrow ILR \oplus AIR$

A_4	A_3	A_2	A_1	A_0

Active Interrupt Register

Clears on Write to AIR

Interrupt Priority Select

IP = 0 No Priority

IP

IP = 1 Interrupts Prioritized

FUNCTIONAL DESCRIPTION

1. IP = 0 No Priority

All interrupt information latched into interrupt latch register (ILR) is immediately transferred into active interrupt register (AIR) and $\overline{IRQ}$ is pulled low. Upon read of interrupt the $\overline{IRQ}$ is reset high and the appropriate bit(s) of the interrupt latch register is cleared by exclusive OR-ing. The ILR with AIR (ILR$\oplus$AIR). After the appropriate interrupt request has been serviced a Write to the AIR will clear it and initiate a new interrupt sequence if any interrupts were received during previous interrupt servicing In this non-prioritized mode it is possible for two or more interrupts to occur simultaneously and be transferred to the AIR. If this occurs it is a software effort to recognize this and act accordingly.

2. IP = 1 *Interrupts Prioritized*

In this mode the Interrupt Inputs are prioritized in the following order $I_4 > I_3 > I_2 > I_1 > I_0$

In this mode only one bit of the AIR can be set at any one time. If an interrupt occurs it is latched into the interrupt latch register, the $\overline{IRQ}$ line is pulled low and the appropriate bit of the AIR is set. To understand fully the operation of the priority interrupts it is easiest to consider the following examples.

A. The first case is the simplest. A single interrupt occurs and the processor can service it completely before another interrupt request is received.
    1. Interrupt I_1 is received.
    2. Bit I_1 is set high in Interrupt Latch Register.
    3. $\overline{IRQ}$ is pulled low.
    4. A_1 is set high.
    5. Processor recognizes $\overline{IRQ}$ and reads AIR to determine which interrupt occurred.
    6. Bit I_1 is reset and $\overline{IRQ}$ is reset to high.
    7. Processor Services Interrupt and signals completion of Service routine by writing to AIR.
    8. A_1 is reset low and interrupt sequence is complete.

B. The second case occurs when an interrupt has been received and a higher priority interrupt occurs. (See Note)
    1. Interrupt I_1 is received.
    2. Bit I_1 is set high on the Interrupt Latch Register.
    3. $\overline{IRQ}$ is pulled low and A_1 is set high.

4. Processor recognizes $\overline{IRQ}$ and reads AIR to determine which interrupt occurred.

5. Bit I_1 is reset and $\overline{IRQ}$ is reset high.

6. Processor begins servicing I_1 interrupt and the I_2 interrupt is received.

7. A_2 is set, A_1 is reset low and $\overline{IRQ}$ is pulled low.

8. Processor has not yet completed servicing I_1 interrupt so this routine will be automatically stacked in 6500 stack queue when new $\overline{IRQ}$ for I_2 of interrupt is received.

9. Processor reads AIR to determine I_2 interrupt occurrence and bit I_2 of interrupt latch is reset.

10. Processor services I_2 interrupt, clears A_2 by writing AIR and returns from interrupt. Returning from interrupt causes 650X processor to resume servicing I_1 interrupt.

11. Upon clearing A_2 bit in AIR, the A_1 bit will not be restored to a one. Internal circuitry will prevent a lower priority interrupt from interrupting the resumed I_1.

C. The third case occurs when an interrupt has been received and a lower priority interrupt occurs.

1. Interrupt I_1 is received and latched.

2. $\overline{IRQ}$ is pulled low and A_1 is set high.

3. Processor recognizes $\overline{IRQ}$ and reads AIR to determine that I_1 interrupt occurred.

4. Processor logic servicing I_1 interrupt during which I_0 interrupt occurs and is latched.

5. Upon completion of I_1 interrupt routine the processor writes AIR to clear A_1 to signal 6525 that interrupt service is complete.

6. Latch I_0 interrupt is transferred to AIR and $\overline{IRQ}$ is pulled low to begin new interrupt sequence.

NOTE: It was indicated that the 6525 will maintain Priority Interrupt information from previously serviced interrupts.

This is achieved by the use of an Interrupt Stack. This stack is pushed whenever a read of AIR occurs and is pulled whenever a write to AIR occurs. It is therefore important not to perform any extraneous reads or writes to AIR since this will cause extra and unwanted stack operations to occur.

The only time a read of AIR should occur is to respond to an interrupt request.

The only time a write of AIR should occur is to signal the 6525 that the interrupt service is complete.

J

Disk User's Guide

The optional disk holds programs supplementary to the book.

The programs are as follows:

SUPERMON1 (for original ROM PET computers)
SUPERMON2(for upgrade ROM PET/CBM computers)
SUPERMON4 (for 4.0 PET/CBM computers)
SUPERMON.V (for VIC-20 computers)
SUPERMON64 (for Commodore 64 computers)
SUPERMON INST (instructions, BASIC)
UNICOPY64 (for Commodore 64)
UNICOPY INST (instructions, BASIC)
]UNICOPY LIST (BASIC, all machines)
UNICOPY ASSY (data file for UNICOPY LIST)
COPY-ALL (for PET/CBM)
COPY-ALL.64 (for Commodore 64)
CROSS REF (for PET/CBM)
CROSS REF 64 (for Commodore 64)
FACTORS (for PET/CBM)
FACTORS V64 (for VIC-20, Commodore 64, and PLUS/4)
PENTOMINOS INST (instructions)
PENTOMINOS (BASIC, all machines)
PENTOMINOS PET (for PET/CBM)
PENTOMINOS V64 (for VIC-20, Commodore 64, and PLUS/4)

PENTOMINOS B128 (boot for B128 system)
+PENTO128 (program for B128)
+XFER (transfer sequence for B128)
STRING THING (BASIC, for PET/CBM)
STRING THING V64 (BASIC, for VIC-20, Commodore 64)
]SAMPLE FILE (for use with STRING THING)

These programs are public domain, and may be obtained from user groups. They are available here for user convenience.

The following notes may be useful in using or studying the programs.

SUPERMON1 (for original ROM PET computers)
SUPERMON2 (for upgrade ROM PET/CBM computers)
SUPERMON4 (for 4.0 PET/CBM computers)
SUPERMON.V (for VIC-20 computers)
SUPERMON64 (for Commodore 64 computers)
SUPERMON INST (instructions, BASIC)

Supermon 2 and 4 are "extensions" to the built-in MLM of the respective machines. The other Supermon versions are complete monitors.

Remember that the programs on disk are "monitor generators," that is, they build the monitor for you. After the monitor has been built, you should remove the builder program so that you don't end up with two copies. In other words, after RUN type .X to return to BASIC, NEW to scrap the builder, and then SYS4 or SYS8 to return to the monitor whenever desired.

The monitor is always built near the top of memory. Its entry address can be determined by checking the TOM (top-of-memory) pointer. Monitors are complex, but feel free to ask the monitor to disassemble itself for your information.

After Supermon is placed, you may load BASIC programs and use the computer normally. Supermon will remain until you shut off the power.

UNICOPY64

A utility for copying files from one disk to another, on a single drive; or copying from one disk to cassette tape. The program is written entirely in machine language, apart from the SYS that starts it up.

Information is copied from the selected files into RAM memory. When the output phase begins, the data is then written to disk or tape.

UNICOPY INST

A BASIC program explaining how to use UNICOPY64.

UNICOPY LIST
]UNICOPY ASSY

An assembly listing of program UNICOPY. Because UNICOPY is written entirely in machine language, a number of tasks are performed in the program that are often more conveniently done in BASIC. For example, files are opened and closed by machine language. This makes the program listing particularly interesting for students of these techniques.

Assembly listings have a somewhat different appearance from the machine language programs this book has dealt with. The most visible difference is in the use of symbolic addresses. If there is any confusion, concentrate on the machine language half of the listing; that will clarify what's going on. Program UNICOPY LIST allows output to the screen or to a Commodore printer.

For cassette tape output, direct calls to the ROM routines are made; that's usually not good practice, but there's little choice here.

The program is written in machine language so that the BASIC ROM can be flipped out, allowing for more memory space in which to copy programs.

COPY-ALL (for PET/CBM)
COPY-ALL.64 (for Commodore 64)

A utility for copying files from one disk drive to another. You will find two SYS commands in the BASIC part of the program: one to get the directory, and the other to do the actual copying.

Information is copied from the selected file into a BASIC string that has been set aside for the purpose. A similar technique may be found in the simpler STRING THING.

CROSS REF (for PET/CBM)
CROSS REF 64 (for Commodore 64)

This program prepares a cross-reference listing for any selected BASIC program on disk. It cross-references both line numbers and variables. It's a good way to document a BASIC program.

The program uses two table lookup techniques that may be confusing to the beginning machine language program reader. First, it classifies all characters received from BASIC in terms of "type"; this is done with a table of 256 elements, one for each possible character. Second, it uses a "state transition table" to record the nature of the job in progress; for example, after meeting a GOSUB "token," it will expect to receive a line number.

The second SYS in the BASIC program is used to print the line numbers of the cross-reference. It employs an efficient binary-to-decimal conversion technique, which uses decimal mode.

FACTORS (for PET/CBM)
FACTORS V64 (for VIC-20, Commodore 64, and PLUS/4)

This program finds factors of numbers up to nineteen digits long. This shows a powerful feature of machine language as opposed to BASIC· the size of numbers is not limited by the language.

The program contains a number of useful techniques worth studying. First, it allows decimal input of any number up to 19 digits (a 64-bit or 8-byte binary number). Second, to find factors it performs division with remainder. Finally, to print results, it must convert binary-to-decimal, using the same decimal mode technique as in CROSS REF

The program does not try all divisors. After trying a few initial values (2, 3. and 5). it switches to a "30-counter" technique, trying only multiples of 30 plus 1, 7, 11, 17, 19, 23, and 29.

The machine language program is relocated by BASIC so that it starts at hexadecimal 1300 regardless of where it was originally loaded. This was originally done to allow for the VIC-20's variable start-of-BASIC, which rambles according to the amount of extra memory fitted. It turns out to be useful for studying to have the program in a fixed location; so the PET/CBM version was also set up in this way.

Students wishing to disassemble FACTORS will find the following information useful:

VARIABLES:

    $0349—number of times a factor divides evenly
    $034A—"equals" or "asterisk" character for formatting
    $034B—zero suppression flag
    $034C—30-counter
    $0350 to $0357—value under analysis
    $0358 to $035F—value work area
    $0360 to $0367—"base" value for 30-counter
    $036C to $0379—division work area, including:
    $036C to $036F—remainder
    $0370 to $0377—quotient

PROGRAM.

    $1300: Main routine, including:
    $1300: Start, clear work area
    $131D: Get number digits from user

$1331: Handle bad input
$133A: Begin factoring; check non-zero
$1350: Try divisors 2, 3, and 5
$1365: Try higher divisors
$13A2: Print remaining value.
$13BA: Prompt subroutine
$13C4: Input and analyze digit
$140B: Multiply-by-two subroutine
$1415: Division subroutine
$147A: Try a divisor (short)
$147D: Try a divisor (long)
$1485: Check if remainder zero
$1492: Log factor if found
$14A2: Check if more to do
$14B9: Print value subroutine
$14D0: Print factor subroutine
$1504: Clear output area
$150F: Convert to decimal and print
$1535: Print a digit with zero suppression
$1565: 30-count values: 1, 7,.11, etc.

Even at machine language speeds, this program can take a long time to analyze large factors and prime numbers. The RUN/STOP key is active to allow the user to stop the run.

PENTOMINOS INST (instructions)
PENTOMINOS (BASIC, all machines)
PENTOMINOS PET (for PET/CBM)
PENTOMINOS V64 (for VIC-20, Commodore 64, and PLUS/4)
PENTOMINOS B128 (boot for B128 system)
  +PENTO128 (program for B128)
  +XFER (transfer sequence for B128)

A puzzle solving problem. Pieces are fitted into a selected rectangular shape "visibly"—in other words, they may be seen on the screen as they are tried.

The machine language programs follow the logic of the BASIC program precisely. The "shape tables" have been rearranged for greater machine language convenience (each piece is reached by indexing; the index range of 0 to 255 dictates the piece being selected and its rotation).

The machine language program uses no indirect addressing and no subroutines. That is not necessarily good practice; it is largely a result of writing the program logic to exactly match the BASIC program.

This program makes use of tables, and is worth studying for that reason. It is also useful to examine the close relationship between the BASIC program and its machine language equivalent, especially starting at line 2000 in BASIC.

As with FACTORS, the machine language program is relocated by BASIC so that it starts at hexadecimal 156D (with tables starting at $12FA) regardless of where it was originally loaded. Again, this is necessary for the VIC-20 and proves to be convenient for study purposes on all machines— except the B-128 version, where this relocation does not happen.

Students wishing to disassemble PENTOMINOS will find the following information useful:

VARIABLES:

$033C—piece number, BASIC variable P
$033D to $033E—variables W1 and W2, board size
$033F—P1, number of pieces placed
$0340 to $034B—U(..) log of pieces placed
$034C to $0357—T(..) rotation of piece
$0358 to $035C—X(..) location of piece
$035D to $0361—Y(..) location of piece
$0362 to $0370—tables to place a piece
$037F to $039C—board "edge" table
$039D to $03D8—B(...) the board.

PROGRAM:

$156D: Start, BASIC line 1070
$15A4: Clear screen, BASIC line 1120
$15A9: Clear variables, set up
$15CC: Find space, BASIC line 2010
$1600: Get new piece, BASIC line 2030
$1609: Try piece, BASIC line 2060
$1686: Put piece in, BASIC line 2120
$16E0: Print "Solution", BASIC line 2170
$1701: Undraw piece, BASIC line 2190
$17AB: Rotate piece, BASIC line 2260
$17BC: Give up on piece, BASIC line 2280
$17C1: Look for new piece, BASIC line 2300

The B128 version does not align to the above addresses. It is written to illustrate the "boot" loading system needed for that computer. Programs whose names begin with a + symbol are loaded by the bootstrap program; do not try to load them directly.

STRING THING (BASIC, for PET/CBM)
STRING THING V64 (BASIC, for VIC-20, Commodore 64, PLUS/4)
SAMPLE FILE

A simple machine language program, POKEable directly from BASIC, to substitute for an INPUT# statement.

INPUT# has several limitations that sometimes make it awkward for use with files:

- No more than 80 characters may be read.
- The comma or colon character will break up input
- Leading spaces will disappear

STRING THING reads everything up to the next RETURN or end of file. It is pure BASIC, but POKEs machine language into the cassette buffer area. It finds the first variable and uses it as an input buffer.

Glossary

The numbers in parentheses indicate the chapter in which the word or phrase is first used.

Absolute address: (5) An address that can indicate any location in memory.

Accumulator: (3) The A register; the register used for arithmetic.

Address bus: (1) A bus that signals which part of memory is wanted for the next memory operation

Address mode: (5) The manner in which an instruction reaches information within memory.

Address: (1) The identity of a specific location within memory

Algorithm: (1) A method or procedure to perform a computing task

Arithmetic shift or rotate: (4) A shift or rotate that usually preserves the sign of a number.

Assembler: (2) A program that assembles or changes source code into object code.

Assembly: (1) The process of changing source code into object code.

Assembly code: (1) Also called source code A program written in a somewhat human-readable form. Must be translated ("assembled") before use.

317

Assembly language: (1) The set of instructions, or language, in which a source program must be written before assembly.

Binary: (1) Something that has two possible states; a number based on digits, each of which has two possible states.

Bit: (1) A binary digit; the smallest element of information within a computer.

Bootstrap: (6) A program that starts up another program.

Breakpoint: (8) A location where the program will stop so as to allow checking for errors.

Bug: (8) An error within a program.

Bus: (1) A collection of wires connecting many devices together.

Byte: (1) Eight bits of information grouped together; the normal measure of computer storage.

Calling point: (2) The program location from which a subroutine is called into play; the subroutine will return to the calling point when finished.

Channel: (8) A path connecting the computer to one of its external devices.

Comment: (8) A program element which does not cause the computer to do anything, used as advice to the human program reader.

Commutative: (3) A mathematical operation that works both ways, e.g., $3 + 4$ gives the same result as $4 + 3$.

Control bus: (1) A bus that signals timing and direction of data flow to the various connected devices.

Data bus: (1) A bus used to transfer data between memory and the microprocessor.

Debugging: (8) Testing a program to uncover possible errors.

Decimal: (1) A number system based on a system of ten digits; the "normal" numbering system used by humans.

Decrement: (2) To make smaller by a value of one.

Descriptor: (6) A three-byte set of data giving a string's length and its location.

Disassembler: (2) A program that changes object code into assembly code to allow inspection of a program.

Disassemble: (2) To change object code into assembly code. Similar to a LIST in BASIC.

Dynamic string: (6) A string that must be placed into memory after being received or calculated.

Effective address: (2) The address used by the processor to handle data when executing an instruction. It may differ from the instruction address (or "operand") because of indexing or indirect addressing.

Event flag: (7) A flag that signals that some event has happened.

Execute: (1) To perform an instruction.

File: (8) A collection of data stored on some external device.

Flag: (3) An on/off indicator that signals some condition.

Floating accumulator: (7) A group of memory locations used by BASIC to perform calculations on a number.

Garbage collection: (6) A BASIC process in which active strings are gathered together and inactive strings are discarded. On some computers this can be quite time consuming.

Increment: (2) To make larger by a value of one.

Index: (2) To change an address by adding the contents of an index register.

Index register: (2) The X or Y registers, which may be used for changing effective addresses.

Indirect address: (5) An addressing scheme whereby the instruction contains the location of the actual address to be used; an address of an address.

Instruction: (1) An element of a program that tells the processor what to do.

Interrupt: (1) An event that causes the processor to leave its normal program so that some other program takes control, usually temporarily.

Interrupt enable register: (7) A location within an IA chip that determines whether or not a selected event will cause an interrupt.

Interrupt flag: (7) A signal within the IA indicating that a certain event has requested that an interrupt take place.

Interrupt flag register: (7) A location within the IA where interrupt events can be detected and turned off if desired.

Interrupt source: (7) The particular event that caused an interrupt. Since many things can do this, it's usually necessary to identify the specific source of the interrupt.

Kernal: (2) Commodore's operating system.

Label, symbolic address: (8) A name identifying a memory location.

Latch: (7) A flag that "locks in."

Load: (1) To bring information from memory into the processor. A load operation is a copying activity; the information still remains in memory.

Logical file number: (8) The identity of a file as used by the programmer.

Logical operator: (3) An operation that affects individual bits within a byte: AND, ORA, and EOR.

Logical shift or rotate: (4) A shift that does not preserve the sign of assigned number.

Machine code: (1) Instructions written in machine language.

Machine language: (1) The set of commands that allow you to give instructions to the processor.

Machine language monitor: (1) A program that allows communication with the computer in a manner convenient for machine language programming.

Memory: (1) The storage used by a computer; every location is identified by an address.

Memory mapped: (1) Circuits that can be reached by the use of a memory address, even though they are not used for storage or memory purposes.

Memory page: (5) A set of 256 locations in memory, all of whose addresses have the same "high byte."

Microcomputer: (1) A computer system containing a microprocessor, memory, and input/output circuits. A computer built using microchips.

Microprocessor: (1) The central logic of a microcomputer, containing logic and arithmetic. A processor built on a microchip.

Monitor: (1) A program that allows the user to communicate with the computer. Alternatively, a video screen device.

Non-maskable interrupt, NMI: (7) A type of interrupt that cannot be disabled.

Non-symbolic assembler: (2) An assembler in which actual addresses must be used.

Object code: (1) The machine language program that will run in the computer.

Octothorpe: (2) Sometimes called a numbers sign, a pounds sign, a hash mark. The "#" symbol.

Operand: (1) The part of an instruction following the op code that usually signals where in memory the operation is to take place.

Operating system: (1) A set of programs with a computer that takes care of general work such as input/output, timing, and so on

Operation code, op code: (1) The part of an instruction that says what to do.

Overflow: (3) Condition caused by an arithmetic operation generating a result that is too big to fit in the space provided.

Pointer: (6) An address held in memory, usually in two bytes.

Processor status word, status register: (3) A processor register that holds status flags.

Pull: (7) To take something from the stack.

Push: (7) To put something on the stack.

Random access memory, RAM: (1) The part of a computer's memory where information can be stored and recalled.

Read: (1) To obtain information from a device.

Read only memory, ROM: (1) The part of a computer's memory where fixed information has been stored. New information cannot be stored in a ROM; it is preprogrammed.

Register: (1) Location within a processor where information can be held temporarily.

Screen editing: (1) The ability to change the screen of a computer and cause a corresponding change in memory

Screen memory: (2) The part of a computer holding the information displayed on the screen. Changing screen memory will change the screen; reading screen memory will reveal what is on the screen.

Selected: (1) A chip or device that has been signaled to participate in a data transfer. If the chip or device has not been selected, it will ignore data operations.

Self-modifying: (7) A type of program that changes itself as it runs. Rare, and not always considered good programming practice

Signed number: (3) A number that holds a value that may be either positive or negative.

Source code: (1) Instructions written in assembly language; usually, the first code written by the programmer before performing an assembly.

Stack: (7) A temporary, or "scratch pad," set of memory locations.

Status register, processor status word: (3) Within the processor, a register that holds status flags.

Store: (1) To transfer information from the processor to memory. The store operation is a copying activity. the information still remains in the processor.

Subroutine: (2) A set of instructions that can be called up by another program

Symbolic address, label: (7) A name identifying a memory location.

Symbolic assembler: (2) An assembler in which symbolic addresses may be used. This is more powerful than a non-symbolic assembler.

Testable flag: (3) A flag that can be tested by means of a conditional branch instruction.

Two's complement: (3) A method of representing negative numbers. With single byte numbers, -1 would be represented by \$FF.

Unsigned number: (3) A number that cannot have a negative value.

Write: (1) To send information to a device.

Zero page: (5) The lowest 256 locations in memory. Locations whose addresses begin with hexadecimal \$00. . .

The six major Kernal subroutines·

Addrs	Name	Action	Registers Changed
$FFC6	CHKIN	Switch input to logical file X	A,X
$FFC9	CHKOUT	Switch output to logical file X	A,X
$FFCC	CLRCHN	Restore input/output defaults	A,X
$FFD2	CHROUT	Output ASCII character in A	none
$FFE1	STOP	Check RUN/STOP key	A
$FFE4	GETIN	Get ASCII character into A	A,X,Y

Some Important Memory addresses (hexadecimal):
Original ROM PET not included.

Pointers.	PET/CBM	VIC	C64
Start of Basic	0028	002B	002B
Start of Variables	002A	002D	002D
End of Arrays	002E	0031	0031
Bottom of Strings	0030	0033	0033
Top of Memory	0034	0037	0037
Status word ST	0096	0090	0090
USR jump	0000	0000	0310
CHRGET subroutine	0070	0073	0073
Floating accumulator	005E	0061	0061
Keytouch register	0097	00CB	00CB